A FAMILY OF SIMULATION PROGRAMS FOR IC FABRICATION PROCESSES (THEIR STRUCTURE, DESIGN, AND IMPLEMENTATION)

by

S. N. Nandgaonkar

Memorandum No. UCB/ERL M84/90

SAMPLE Report No. SAMD-10

22 October 1984

A FAMILY OF SIMULATION PROGRAMS FOR IC FABRICATION PROCESSES (THEIR STRUCTURE, DESIGN AND IMPLEMENTATION)

by

S. N. Nandgaonkar

SAMPLE Report No. SAMD-10

Memorandum No. UCB/ERL M84/90

22 October 1984

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

A Family of Simulation Programs for IC Fabrication Processes (Their Structure, Design, and Implementation)

By

Sharad Narayan Nandgaonkar

B. Tech (Indian Institute of Technology, Bombay, India) 1975
M.S. (University of California) 1978

DISSERTATION

Submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

Computer Science

in the

GRADUATE DIVISION

OF THE

UNIVERSITY OF CALIFORNIA, BERKELEY

Approved: William G. Oldham 10/22/84
Chairman Date

Andrew R. Neureuther

Yiell A. Bakru

A Family of Simulation Programs for IC Fabrication Processes (Their Structure, Design, and Implementation)

by

Sharad Narayan Nandgaonkar

4.2BSD stands for the Berkeley Software Distribution version 4.2 of the Unix system from the Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, California U.S.A.

Calidoscope stands for CAL's (UC Berkeley's) Improved Design of SCOPE (the SCOPE operating system of CDC).

CDC is a trademark of the Control Data Corporation, U.S.A. (CDC 6400)

DEC, PDP, RSX, VAX are trademarks of the Digital Equipment Corporation, U.S.A. (DEC-10, PDP 11/40, PDP 11/70, RSX-11M, VAX 11/780)

HP is a trademark of the Hewlett-Packard Corporation, U.S.A. (HP1000, RTE-4B)

Unix is a trademark of AT&T Bell Laboratories, U.S.A.

□

A Family of Simulation Programs for IC Fabrication Processes (Their Structure, Design, and Implementation)

Sharad Narayan Nandgaonkar

Ph.D. Dissertation

Department of Electrical Engineering and Computer Sciences
University of California, Berkeley, California 94720 U.S.A.

Abstract

Simulation is a powerful technique of engineering research. A methodology for the design, construction and analysis of software to simulate many related processes, both individually and in sequences, is presented. The experience from the SAMPLE software project to simulate Integrated Circuit fabrication processes is distilled in the form of principles, models and guidelines of general applicability.

The goal of simulation is characterized as the substitution of real entities by computations. This motif of imitation of reality is stated as a general principle: the matching of computational resources to simulation needs and desires. The design and construction of software to realize this matching is guided by a systematic mapping between them through traditional intermediate stages like physical modelling and numerical methods. A state-variable model is used to maintain uniformity of structure and interaction at all levels in that mapping. The process sequences are represented by a diagram equally applicable in the real laboratory as well as for the simulation software. The user-interaction with the software via input and output is similarly structured to obtain a uniform handling for all processes.

The application of these models and principles to the overall abstract design as well as to various practical details is illustrated using the SAMPLE software as a comprehensive example. Their use has resulted in a coherent structure for the SAMPLE family of simulation components and its documentation. Also shown is their use in analyzing and evaluating the software in its current form, its past development, new ideas for extending it, and in obtaining guidelines for future work. □

William C Oldham

Committee Chairman

Dedicated to
my parents

Acknowledgements

I wish to thank my advisors Professor William G. Oldham and Professor Andrew R. Neureuther whose initiative and support made it possible for me to participate in this project. It is due to their foresight that this project was launched and progressed from its very modest beginnings to achieve the results obtained over all these years. I would also like to thank Professor Kjell A. Doksum for serving on my dissertation committee as an advisor from an external field. With the fewest possible words he reassured me during the qualifying examination research proposal stage and in the final dissertation writing stage.

This project has been a teamwork effort. In particular I wish express my appreciation of the work and the personal friendship extended to me by Michael M. O'Toole, the late John G. Mouton, and Claudio A. Fasce. It has been an enriching experience to work on this project with my fellow students and colleagues. Robert E. Jewett, the late John L. Reynolds, Shankar Subramanian, Chia-Kang Sung, Michael G. Rosenfield, Pradeep K. Jain, George A. Stephens, Frank W. Wise, Wing-Yu Leung, Gino Addiego (with a special flourish), Keunmyung (Ken) Lee, Wendy L. Fong (Wong), Hudy S. Damar, Yoshi Sakai, Albert S. Chen and Ivan Man-Chiu Yeung have all contributed to the project and influenced the work I did on it. I would like to thank them and Guanjung (John) Chern, Yosi Shacham-Diamand, Paul G. Carey and Frandics P. Chan for making life in Berkeley more than just study and work.

From the faculty at UC Berkeley I would like to thank Professors Lotfi A. Zadeh (who was a major influence in getting me interested in the field of Computer Sciences during my first quarter at UC Berkeley), Chakravarthi V. Ravi, Herbert B. Baskin (whose niceness I didn't realize till a fews later), Eugene L. Lawler (who was the other major influence on me during that first quarter), Chittoor V. Ramamoorthy, Lawrence A. Rowe, Alan J. Smith, and Alexandre J. Chorin for encouragement or various other reasons; Chen-Ming Hu for the Teaching Assistantship (of Power Systems Laboratory) that started strictly for money but where I first had the vision of a computer program "just like a set of machines"; and Robert L. Hartman for showing me many ways of looking at things.

I would also like to thank many people from the staff of the EECS Department and the Electronics Research Labortory: among them Teruko Ohashi who earlier this year did some paperwork without having been asked, and Thomas A. King and Boni Kofer for helping me use the drafting facilities for the figures.

It is a pleasure to acknowledge the contributions of William T. Nye, and Supriya Chakrabarti and Joanne Soljack just by being my friends. I also owe special thanks to Connie and Barney Etcheverry, Jean Etcheverry, and Jim and Beth Gillette for being a wonderful Host-family to me in Berkeley. Similarly, many students I met at the Lothlorien "veggie" student Co-op have contributed by making Berkeley a special place for me.

There are a few people to whom I owe more than thanks for the various things they did for me and made my stay in the U.S.A. a memorable experience. Perhaps some day I may find the occasion and the words to convey those feelings to them. Though I am afraid that I may never get a chance to meet some of them again.

This dissertation is dedicated to my parents whose unwavering love and support (in spite of some of my most irresponsible actions and especially when I was so far away and they had very little news) was always a source of strength to me. I do not know of any way to lessen the pain I have caused them by dragging my degree-work out for so long.

The financial support from various Semiconductor Industries in California and from the National Science Foundation is gratefully acknowledged. I hope they have obtained good returns. □

Table of Contents

□

List of Figures

□

Chapter 1

Introduction

§ 1.1 General:

Simulation is a very important tool in modern engineering research and practice. It is one way of tapping the vast potential of the constantly growing power of digital computers to aid in understanding and managing various natural phenomena and ever more ambitious engineering tasks.

In the Integrated Circuit (IC) Electronics field many successful simulation programs have been written over the years to gain insight into all levels of the technology — from IC fabrication (SUPREM [Anto79], ICECREM [Ryss80], SAMPLE [Oldh79, Oldh80]), device structures (MINIMOS [Selb80]) and circuits (SPICE [Nage75]) to the many higher level systems synthesized from them. By their very purpose, these programs are subjected to the demands of simulating more challenging technologies, processes and phenomena, as well as being able to utilize well the computing resources, new hardwares, architectures, peripherals, and software (utilities, languages, operating systems). Under these forces these programs themselves have evolved to become major pieces of software that need a careful analysis of their design and construction.

In this dissertation, the design and structure of the SAMPLE family of simulation programs for IC fabrication processes are discussed from a software project viewpoint. From its early stages the underlying theme in this project has been the imitation of the actions and events in a real laboratory by a computer program (a family of programs as designed now) [Nand78]. Designing the programs to maintain a close correspondence between a real lab and the simulated structures is a good example of applying a more general principle: *Matching the computational resources to the simulation needs and desires.* The simpler the way in which this match can be obtained the simpler, hence more understandable and manageable, will the software be. This often involves approximating the laboratory machines, materials and processes by more tractable models for which code can be written using the available computing resources. The hardware, including the peripherals, for a project usually does not change quickly, although different installations tend to use somewhat different set-ups and tend to have

different usage patterns. The software, on the other hand, can start from a simple set of primitives, and can grow a lot in its sophistication by building different modules and libraries useful for the simulation. Once the software starts doing some of the desired tasks, it can be built upon and refined to a great extent. And the match-point between the simulation tasks and desires, and the software can move up to have more and better simulation capabilities.

§ 1.2 Dissertation Outline

In Chapter 2 a model of the simulation style is presented along with a description of the available computing resources for which the programs were written. Chapter 3 presents the system configuration for the processes that are to be simulated and how they have influenced the overall software structure. Chapter 4 tells about the structure of the individual simulation process components. Chapter 5 tells about the programming aspects, the development, evolution and management of the code. Chapter 6 brings together many points and considerations, including those mentioned in the previous three chapters, to provide a feeling for the tension and balance between them and to provide a practical perspective on this whole software activity. Chapter 7 rounds out the dissertation with a retrospective and outlines of the future directions for the evolution of this project. □

Chapter 2

Simulation Philosophy, Computing Resources

§ 2.1 General

Once some physical processes are made amenable to analytical or computational investigation, programs can be written to aid the investigation.

What is the nature of the programs so generated? Are they a mere collection of random idiosyncratic tools or do they have a completeness in representing the processes they claim to model? From the point of view of a user whose interests are in the physical processes themselves, these characteristics of a program are judged from the information that can be obtained by interacting with it as compared to the information that can be obtained by interacting with the physical processes or entities themselves. This is explicitly stated by a simple input-output model given in the following section.

§ 2.2 The Substitution View of a Program

A user working with some equipment and entities, say wafers, in a physical laboratory (Figure 2.1a) can be considered to be observing the wafers for fabricating ICs, performing some actions on them and observing the effects of those actions on the wafers. This assumes that the user, for the moment at least, is interested only in the *information* to be extracted from the wafers and not the wafers themselves.

If this kind of direct experimentation were replaced by a set-up with a well defined *communication link* through which the desired actions are fed to the experimental apparatus and the results transmitted back to the user we would have a model of this *interaction by information transfer* between the user and the equipment (Figure 2.1b).

Going one step further, the equipment could be substituted by a computer that interprets the commands coming from the user over the communication link and computes, according to some programs already put in it, the effect those commands would have had on the wafers and sends back the computed results (Figure 2.1c). This is the simplest way of specifying a (set of) simulation program(s).

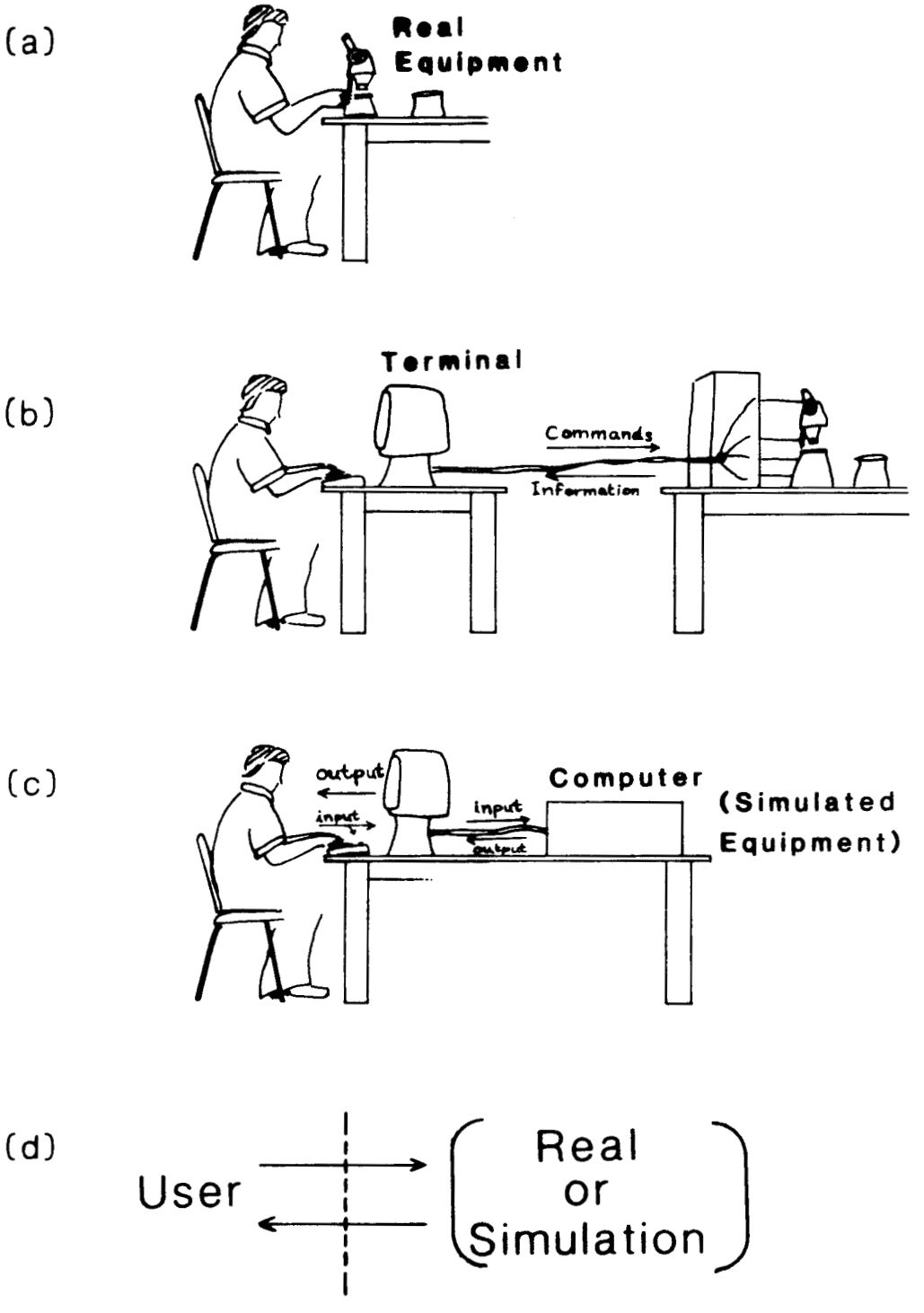


Figure 2.1 The substitution view of a set of simulation programs
a) Direct experimentation
b) Experimentation through an interpreting agent
c) Pure simulation
d) The substitution view

This operational view gives an input-output specification that a set of simulation programs would have to satisfy to be useful in studying the real processes. Ideally, a completely successful simulation will be indistinguishable from a real experiment when the only interaction is via the information transfer over the communication link — barring secondary considerations like speed of response (Figure 2.1d). (This may sound like the Turing Test for intelligence in computers with a laboratory system or process, rather than a human, being simulated by the computer. But unlike in the Turing Test only the behaviour of the physical processes is being considered and not the evaluation of any intelligence.)

Even partial substitution of physical processes by simulated processes may be a desired goal, e.g. when experimental data collection programs are feeding data to a simulation program or when a simulation program is controlling some part of an experiment. This could point out the desirable extensions to be made to the program. For example, there should be a convenient way to enter an experimental wafer profile as a starting profile for the simulation, or to enter experimentally obtained rate curve data for profile advance in the simulation (both capabilities not present in the current version of SAMPLE).

Before discussing other philosophical aspects of the design and implementation of the program (in section 2.4) let us take a look at the computing resources for this project.

§ 2.3 Computing Resources

The computing and peripheral equipment available to the user shapes many of the operational characteristics of the programs. This equipment forms the lowest, though influential, level of the computing resources that the user interface of a program deals with. Other higher level resources, mostly software, may include special-purpose subroutine libraries for graphics output/input, databases for terminal characteristics, special language processors, program development systems, project management aids etc. These higher level resources are useful to the programmer(s) for building the program but are not of direct interest to the end-user.

In the very early stages of development of the SAMPLE program (in 1977) the usage style was shaped by a system with a batch mode of operation (a CDC 6400 computer with the Calidoscope operating system). The input was on a card deck and output on a line-printer (Figure 2.2a). The input could be sent from an interactive system (a PDP 11/70 with the Unix operating system) over a communication line, and the output sent back to the same system; but that didn't significantly change the nature of the batch interaction. Soon another optional output, punched cards containing profile data, was added (by Mike O'Toole) to the program. The punched cards could be taken to a digital pen-plotter with a card-reader to obtain high resolution graphical profile output (Figure 2.2b). This was useful for presentation purposes but for most of the work the the main output was the line-printer printout with textual information and

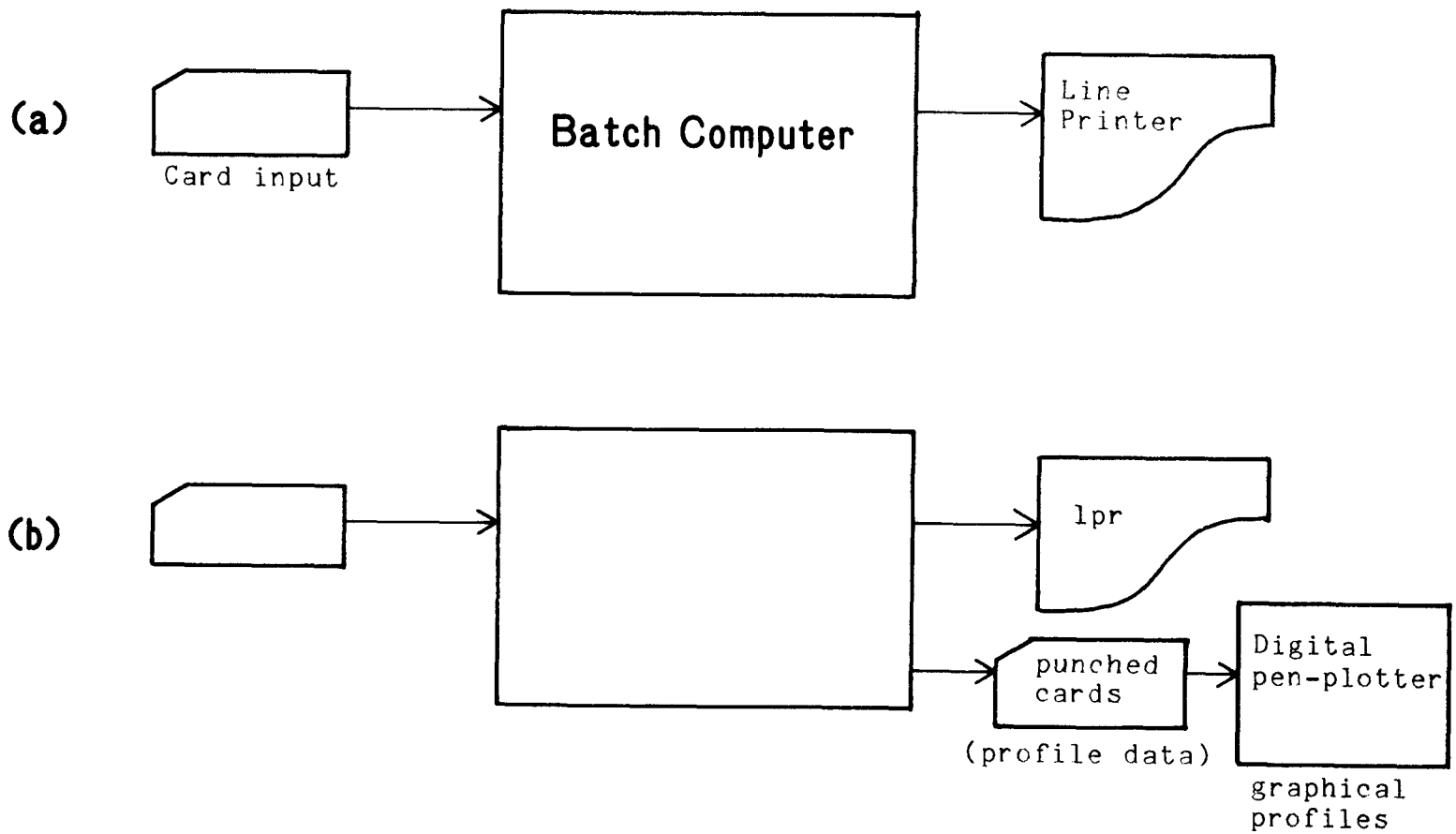


Figure 2.2 An early batch-mode style of interaction with the program
a) A minimal configuration
b) with an off-line plotter

character-array plots.

Then with the availability of a supermini computer (VAX 11/780) with an interactive system (Unix) the program development was much facilitated even though the usage style changed slightly (Figure 2.3a). However no high resolution graphical output was available from this configuration. (Actually graphics terminals were being used in text-only mode because no graphics software was yet written to make use of their graphics capabilities.) Soon a communication link set up between this system and the batch system (of Fig. 2.2) with the card-punch made it possible to use the digital plotter for high-resolution graphical output although it was a little tedious to go through the various links.

The computer site-configuration that was finally found to be comfortably adequate for our purposes is shown in Figure 2.3b. The availability of an acceptably high-resolution graphics terminal (with graphics display software written for it), with an attached unit to get a hardcopy of the display, makes it possible for the programs to convey profile information in a convenient form to the user. Considering that SAMPLE is intended to be used for the study of profiles, this seems to be a minimal comfortable configuration for its usage. Yet in the interest of wide portability and usage there is no built-in dependence in the program code on this configuration. It can display the profiles in the form of character-array plots in the line-printer output. In fact, the students in a course in IC processing techniques at UC Berkeley use the program in essentially the primitive configuration of Figure 2.2a.

For program development and debugging also the configuration of Figure 2.3b has proved adequate. The availability of a magnetic tape drive and computer network connection (not shown in Figure 2.3b) makes it quite convenient to port the software to several other computers as well as to transfer data among them. The versatile programming environment provided by the Unix operating system and its utilities has been a very helpful software resource that has greatly aided the construction of all this software.

§ 2.4 Methodological Considerations

Section 2.2 gave a very general view of simulation itself. For any specific simulation project first the physical system(s) to be simulated will have to be outlined in enough detail so that, at least in principle, the simulation program and the system can be interchanged. For the SAMPLE project an outline of the system to be simulated is given in Chapter 3.

Before getting into those details some general observations should be noted.

First, in Figure 2.1b, a decision has to be made as to which information is to be transferred to the user and which actions the user is allowed to invoke. Due to the complexity of the real processes it is always possible that the decisions made regarding these will have to be

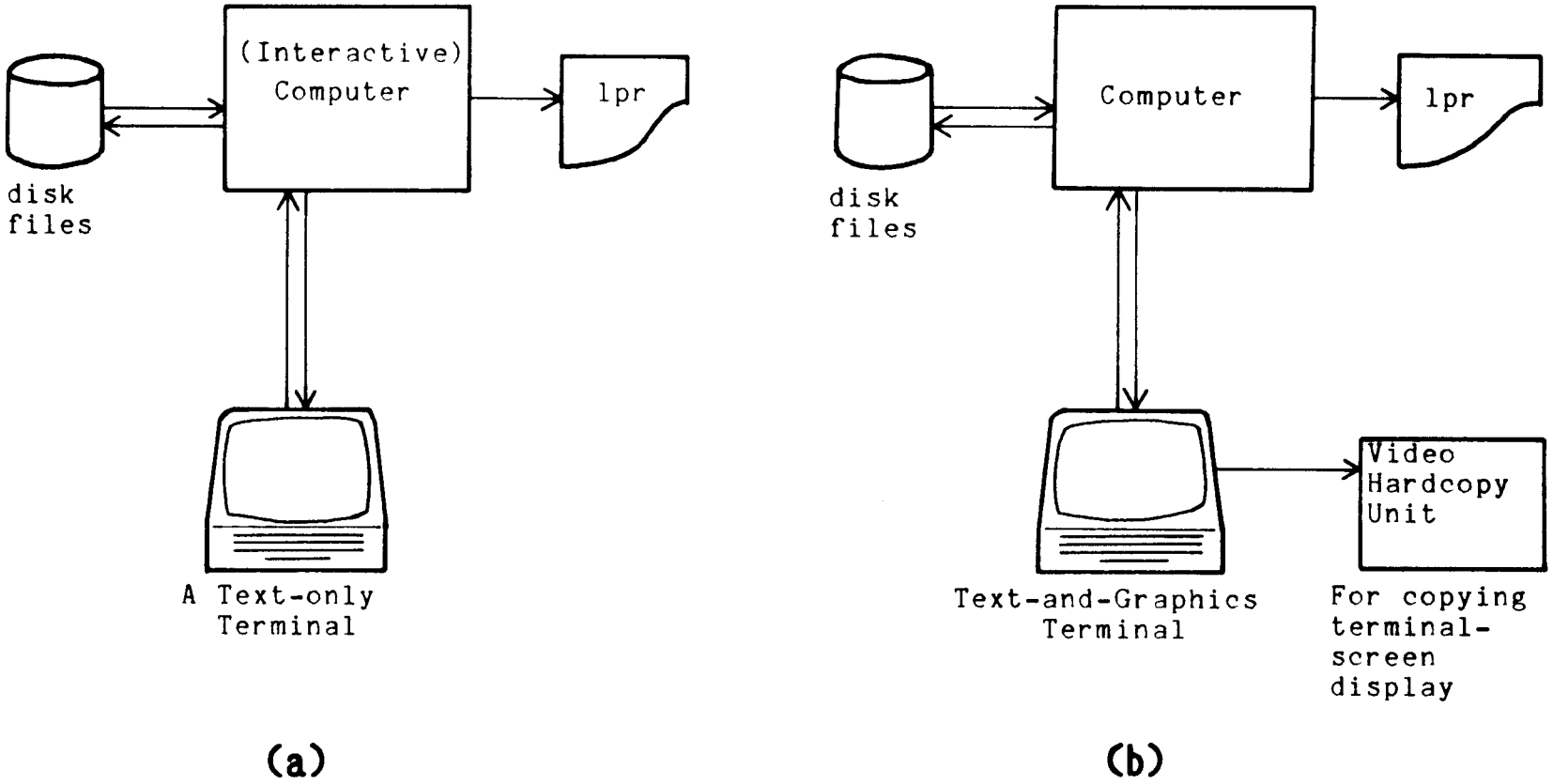


Figure 2.3 Current computer equipment configuration for interaction with the program
a) A minimal configuration
b) A comfortably adequate configuration

modified later on. The program should be flexible and extensible to allow this.

Second, to be flexible and extensible to as yet unplanned aspects of the processes to be studied, the simulation program will have to *imitate* the events and actions in a real laboratory as closely as possible (*the motif of imitation*). Otherwise for the simulation of a slightly modified process or process-sequence a disproportionately large change would be necessary in the simulation program. This would happen when the desired modification violates some assumption made by the program about the process or the process-sequence. For example, in photolithography simulation, if the program relied on an assumption that a wafer development process can only start with an initial flat resist-profile then simulating a two step development with a change of developer after the resist has been developed for a certain time would not be possible without code modification — an unnecessary dependence of the user on the programmer. Such an assumption may easily get built in the program if the code initializes the resist profile at the start of the development process simulation instead of initializing the wafer to that initial profile outside the development simulation part of the code.

Third, due to the complexity of the real processes there may be aspects of the dynamics or details of the processes that are not modelled completely by the program. The degree of this *incompleteness in modelling reality* can only be judged by comparing the output of the program with the results of experiments. For example, if the surface inhibition effect during development is not modelled by the program then outputs from the program will show noticeable qualitative differences near the original resist surface when compared with experimental results.

Fourth, the methods used for simulation may have their own peculiarities that may dominate over the actual effects in the simulated output (somewhat like an information signal being masked by noise in its detector). The most well-known examples of such a possibility of misdirection are the instability effects possible when using some numerical methods for solving continuum problems. Less glamorous examples could be a sloppy choice of parameters for a good numerical method, or simply the insufficient resolution in the graphical output device creating a wrong impression about the accuracy of the results calculated by the program.

The user can maintain a proper perspective on these issues of incompleteness of modelling, and peculiarities of the simulation and computation techniques by keeping in mind the layered structure of the simulation to be presented in Chapter 4. Such a perspective is an invaluable aid in obtaining insight in the results of the simulation or the experiment. It is also very helpful in debugging the code as well as for explanation or justification of correct but apparently counterintuitive outputs from the program. Short of a complete correctness proof, nothing else builds up confidence in the program as much as the finding of a convincing explanation or proof of the validity of some unexpected output from it.

Fifth, because of the computational nature of the simulation it can easily give the user more *controllability* and *observability* in the entities to be studied than is conveniently possible in a laboratory [Nand78]. For example the variation of many parameters of the system is usually far easier with a program that allows it than with a real experimental setup. And there may be no easy nondestructive way of viewing the profiles in time-evolution for a real wafer.

Now in the next chapter we can look at the system being modelled by the SAMPLE program(s) and how it affects the interaction of the program(s) with the user (and with each other). □

Chapter 3

Processing System Configuration

§ 3.1 General

The goal of the SAMPLE project is to study the IC fabrication processes that shape the topography on a wafer. In a typical processing sequence, starting with a flat blank wafer, various planar layers (e.g. oxide) are created on it. A planar layer of a suitable resist is spun on this wafer in preparation for a lithographic operation. The lithographic operation chosen (from optical, e-beam, x-ray or ion-beam lithography) creates the first desired nonplanar topographic pattern in the sensitive resist layer on the wafer. This pattern is further enhanced and transferred to the other layers on the wafer by different etching and deposition steps. The deposition step typically adds a new metal layer to connect different parts of the wafer. By suitable combinations and iterations of etching and deposition steps (Figure 3.1) a complex pattern can be formed on the wafer. In some sophisticated processing sequences even the lithographic operations may be repeated using multilayer resists to control the profiles on the wafer. (These multilayer resist processes are currently outside the simulation capabilities of the SAMPLE software.)

§ 3.2 Processing Sequences

Figure 3.1 shows the possible processing steps that a wafer may be subjected to in a typical processing sequence. That figure is expanded to figure 3.2 to enumerate the different processing steps used in a laboratory. Since the equipments and resists used in the different lithography steps are quite different from each other there is usually no interaction between the various lithography steps. (This is not true for multilayer resist processes, but that does not affect the concept of process sequences presented here.) Any given processing sequence in this system configuration can be represented by a path that the wafer may follow through the different processing machines. This view of the system configuration emphasizes the processing equipment and machines in the laboratory.

Figure 3.3 shows a generalization of this view of the laboratory organization to include as yet unspecified processes that may be added to it. The wafer is passed through the appropriate

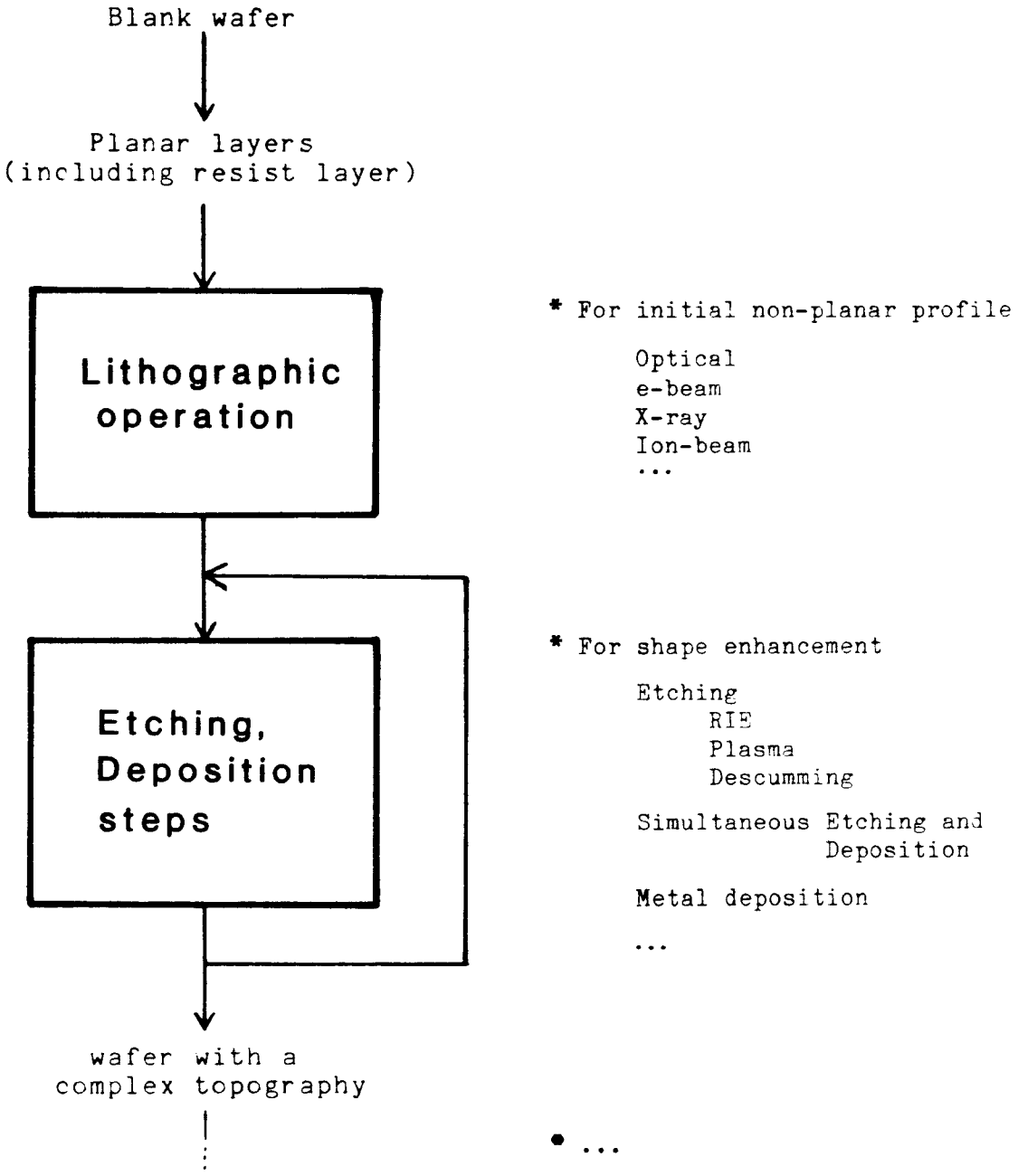


Figure 3.1 IC fabrication processing steps for obtaining desired topographical profiles on a wafer

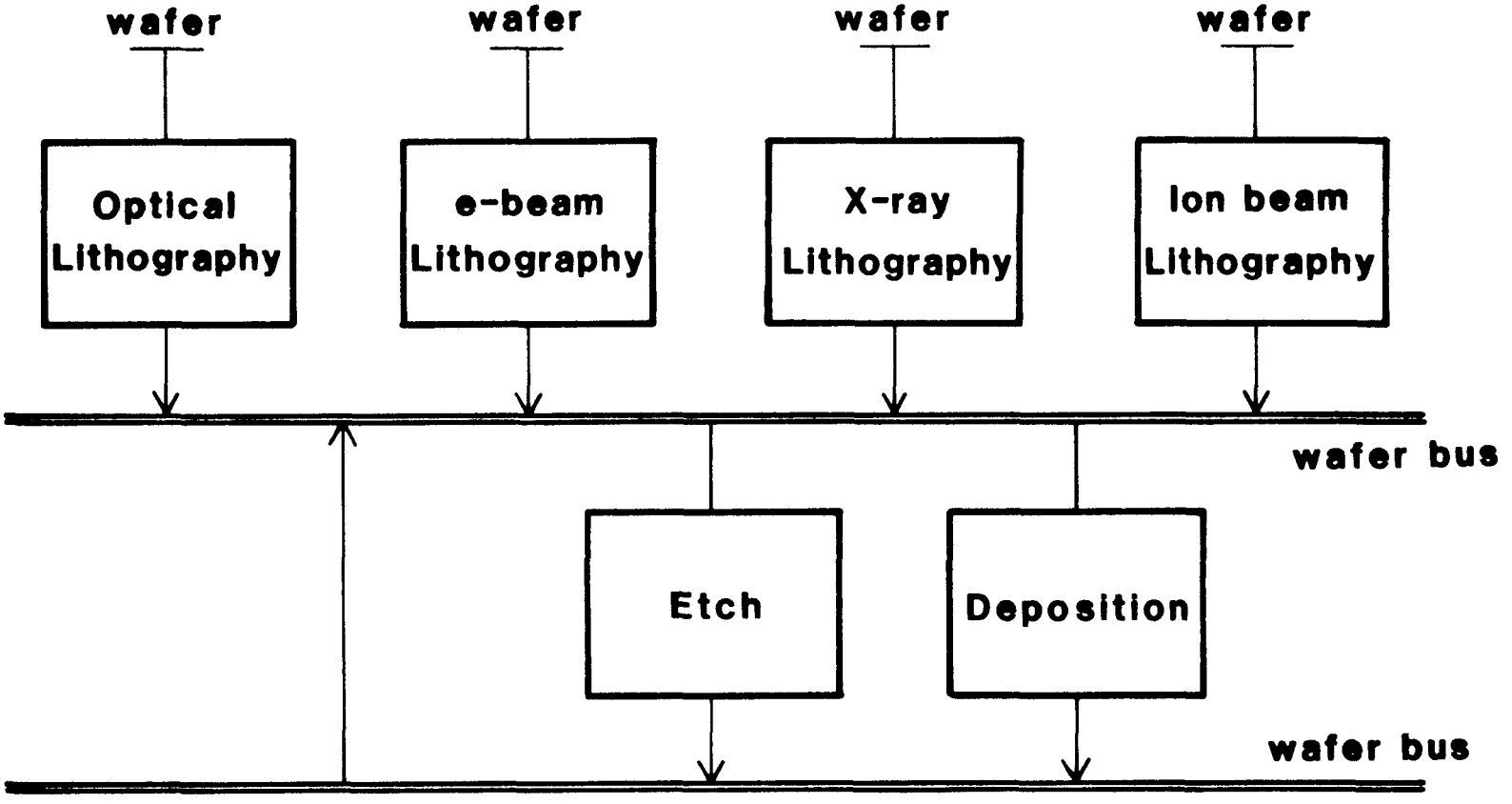


Figure 3.2 Expanded version of Fig. 3.1 to enumerate the different process steps

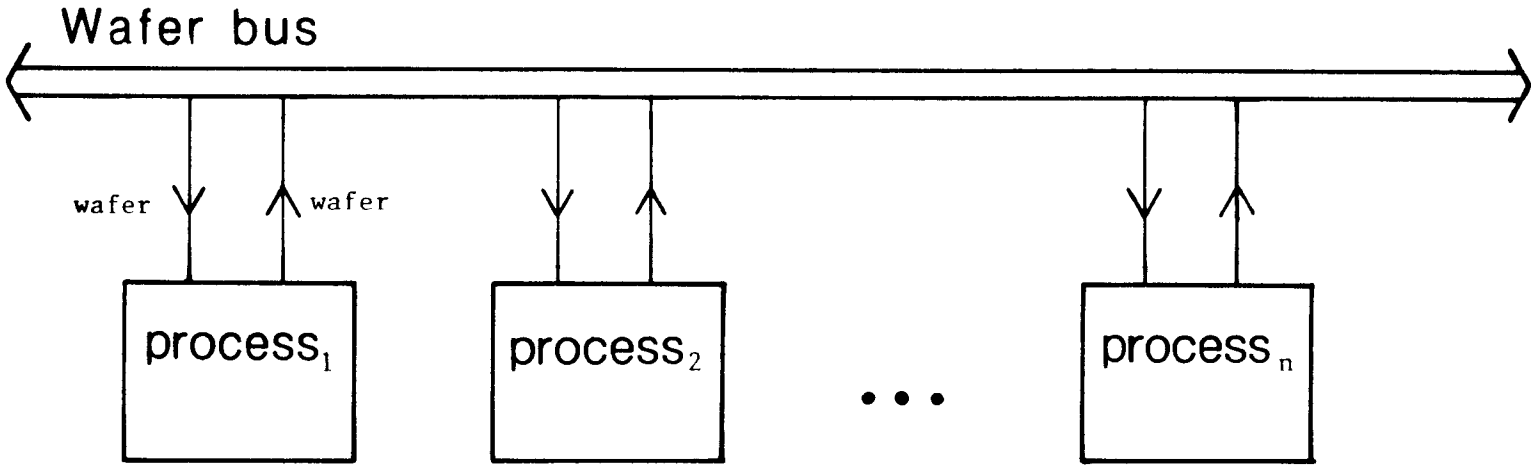


Figure 3.3 Generalized view of the laboratory machine organization to allow arbitrary processing sequences

machines by the user in the order desired to achieve a given processing sequence.

Further generalization of this interaction between wafers and machines can be made by replacing the wafer bus by a collection of wafers that have undergone various steps of processing (Figure 3.4). This view of the system is oriented towards the wafers — with the machines performing the selected operations on the wafers and then returning them back to the collection.

If only one wafer is considered then Fig. 3.4 is essentially equivalent to Fig. 3.3. But the view in Fig 3.4 is more convenient in considering concurrent investigation of different processing sequences which may or may not share some processing steps. This is a common situation in a laboratory when, for example, after generating many wafers with certain common processing steps the effects of varying some parameters of the next process are studied by subjecting the wafers to the different runs of the next process with different values of the control parameter.

In Figures 3.3 and 3.4 the processing machines could simply be for observation (e.g. a microscope) or measurement (e.g. an electrical parameter measuring instrument). In that case their output is the information about the wafer state that they convey to the user (and ideally they would not change the state of the wafer). For theoretical completeness one may even hypothesize an infinite source of blank wafers and an infinite sink of processed wafers as processing machines available in the system.

There is one point not explicitly expanded in Fig. 3.2. Each of the lithographic processing operations is physically two distinct processing steps: a resist exposure step followed by a resist development step (Figure 3.5). The mechanism of exposure varies depending on the type of lithography used. In optical/uv or x-ray lithography a pattern on a mask is imprinted in the resist layer using electromagnetic radiation. In e-beam or ion-beam lithography a pattern is imprinted in the resist by electromagnetically controlling a beam of electrons or ions in a particle gun. The resulting chemical changes in the resist allow the developer used in the development step to selectively remove portions of the resist layer creating the desired geometric profiles in it. Due to the close relation between these two steps in the total lithography operation, many times it is convenient to group them together when considering longer processing sequences. However, since they are physically two distinct steps, it is possible to control each step independently of the other and then they will have to be considered as separate processes within Figures 3.2 and 3.3.

Before we look at the details of each processing step and how they are simulated (Chap. 4) let us consider what the processing sequences themselves imply for the simulation program(s).

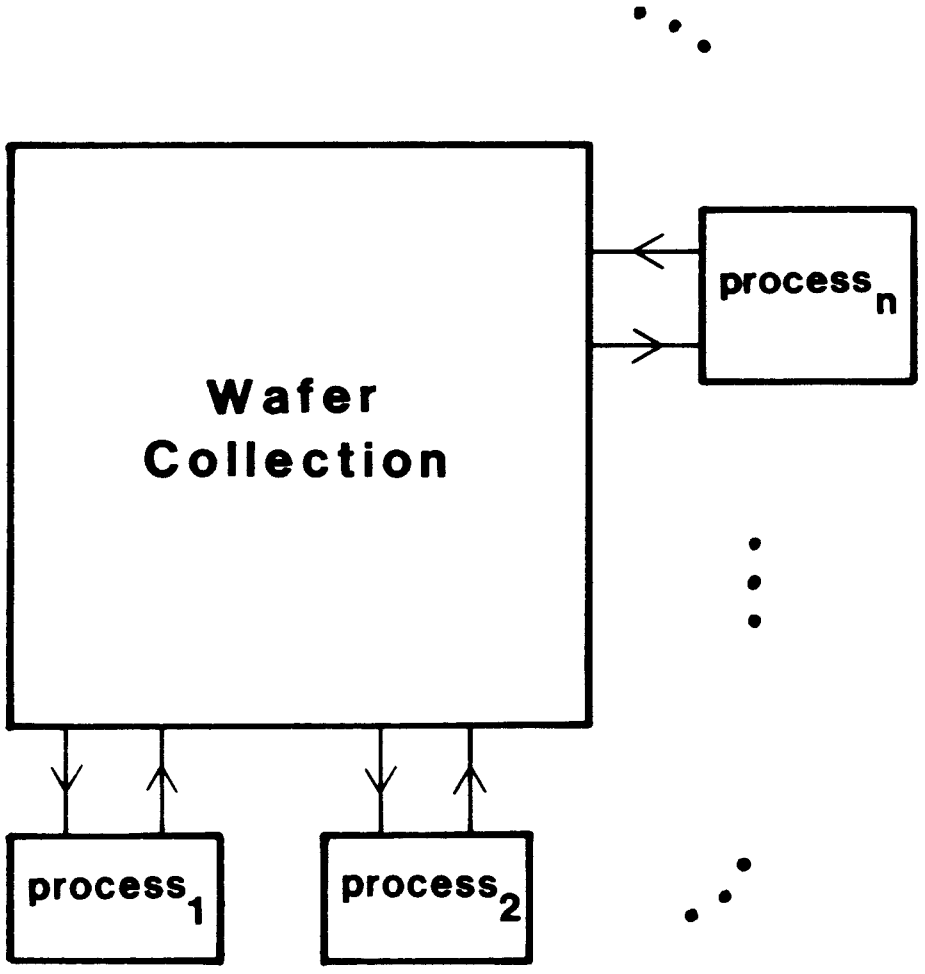


Figure 3.4 Further generalization of Fig. 3.3 emphasizing the wafers

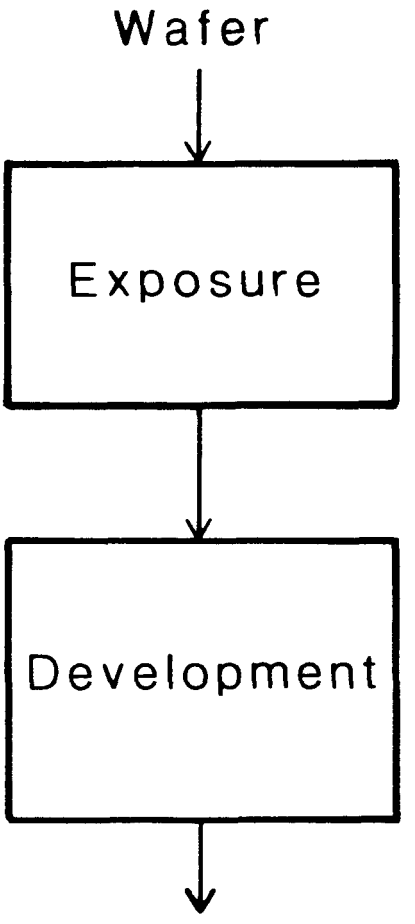


Figure 3.5 The two major steps in each lithographic operation

§ 3.3 Simulation of the Processing Sequences

For simulating any processing step by a program the wafer will have to be represented by data that specifies its physical and chemical states and which can be transformed by routines (simulated machines) that correspond to the processing steps [Nand78]. In this *state-variable representation*, subjecting the wafer to a processing step is simulated by the transformation of data specifying the wafer state by the corresponding simulated machine to give the processed wafer state. Physically taking a wafer from one processing step to another corresponds to *communicating* the wafer data from one simulated machine to another. And by the motif of imitation (§ 2.4) this communication structure should reflect the actual wafer paths through the processing equipment as represented in Figures 3.2, 3.3 or 3.4.

A Note on Some Terminology to be Used Here: Because the SAMPLE software is on the verge of a reorganization (§ 5.10, § 7.4) at present (Summer 1984) some terms to be used in the following discussion need some context and clarification. In all the release versions of the SAMPLE program up to now, the latest being version 1.5b in May 1983, all the routines for the simulation machines were implemented as parts of one computer program. This implementation organization will be referred to as Full-Lab-Simulator (FLS) or the "Together-version" of the program organization. The software is currently being organized as one program per simulated machine. These programs will be referred to as Individual Machine Simulator (IMS) programs or the "separated-version" of the program organization (§ 5.10). The high resolution graphics plot program was always a separate program — first for implementation reasons (§ 2.3) and later for portability and code management reasons. To avoid unnecessary confusion, its separateness will be ignored in the following references to the together-version of the software. Similarly for the Resistance and Capacitance Evaluation program RACPLE [Lee83] which uses the plot-data file to get the profile information from SAMPLE. Note that a plot program corresponds to an observation instrument like a Scanning Electron Microscope (SEM) and the RACPLE program corresponds to an electrical parameter measuring instrument like an impedance-meter. **End of Note.**

In the together-version of the SAMPLE software this communication of the wafer state between the simulated machines was achieved simply by storing the data in a static data structure (Fortran COMMON blocks) accessible to all the machines. This led to only one simulated wafer being present in the system (no copies were made within memory) and the communication between machines was as shown in Figure 3.3. Furthermore, having one static data storage area in the program without any adequate provision to store the wafer state outside the program meant that after invoking a simulation machine the previous state of the wafer was superseded by the new state and hence not available again unless the steps leading up to that point were simulated again.

By contrast, the separated-version of the SAMPLE software as a family of simulation programs communicating data using files (stored on disk memory) corresponds to the more general organization of figure 3.4. The aim of this implementation (with communication through files that store the wafer state) is to allow the user (and the programs) direct access to many of these wafer files (simulated wafers having undergone certain processing steps) at any time after simulating them once. The different simulation machines need not all be written in the same programming language or even be present on the same computer as long as their interface to the wafer-files is well defined.

- = -

In this chapter the interprocess interaction aspects of the simulation were discussed. In the next chapter the intraprocess structure for the simulation of the component processes in Figure 3.2, 3.3 and 3.4 will be considered. □

Chapter 4

Simulation of the Process Steps

§ 4.1 General

The steps in a processing sequence constitute the natural components of the operations in a laboratory to be studied as units. Usually these are considered as single functional steps because of the apparatus used or because a single type of physical phenomenon (or a group of closely related ones) characterizes the actions taking place during that time. These steps could be subdivided further in terms of the detailed procedures and actions needed to operate the equipment, or they may be considered together when their combined result is easier to describe as a unit step than their individual results (e.g. the resist exposure and development steps may be considered together as one major lithographic step). By the motif of imitation (§ 2.4) the simulation program should provide similar relatively modular operations for the user and should be able to accommodate such shifts of viewpoints.

§ 4.2 Structure of the Simulation

A simulation module, a part of a larger program or a separate program, corresponding to a processing step performs the computations corresponding to the actual phenomenon and produces the computational results to be interpreted in the context of the real process. It bridges the real phenomenon to the computing resources by a chain of correspondences, or mappings, as shown in Figure 4.1.

The simulation activity spans a whole range of fields as shown in the figure (Figure 4.1). The layered structure of this activity makes it possible for us to study and understand it in parts (*divide and conquer*). Each layer in the figure is a distinct link of the mapping from one boundary (the physical phenomena under investigation) to the other (the available computing resources). To be meaningful for simulation purposes, each pair of layers has a well-defined correspondence between them. In one direction (from top to bottom in Figure 4.1) it is the *modelling, analysis, or representation and computation* that links them to each other, while in the other direction it is the *interpretation of the results* thus obtained, that complements their relationship.

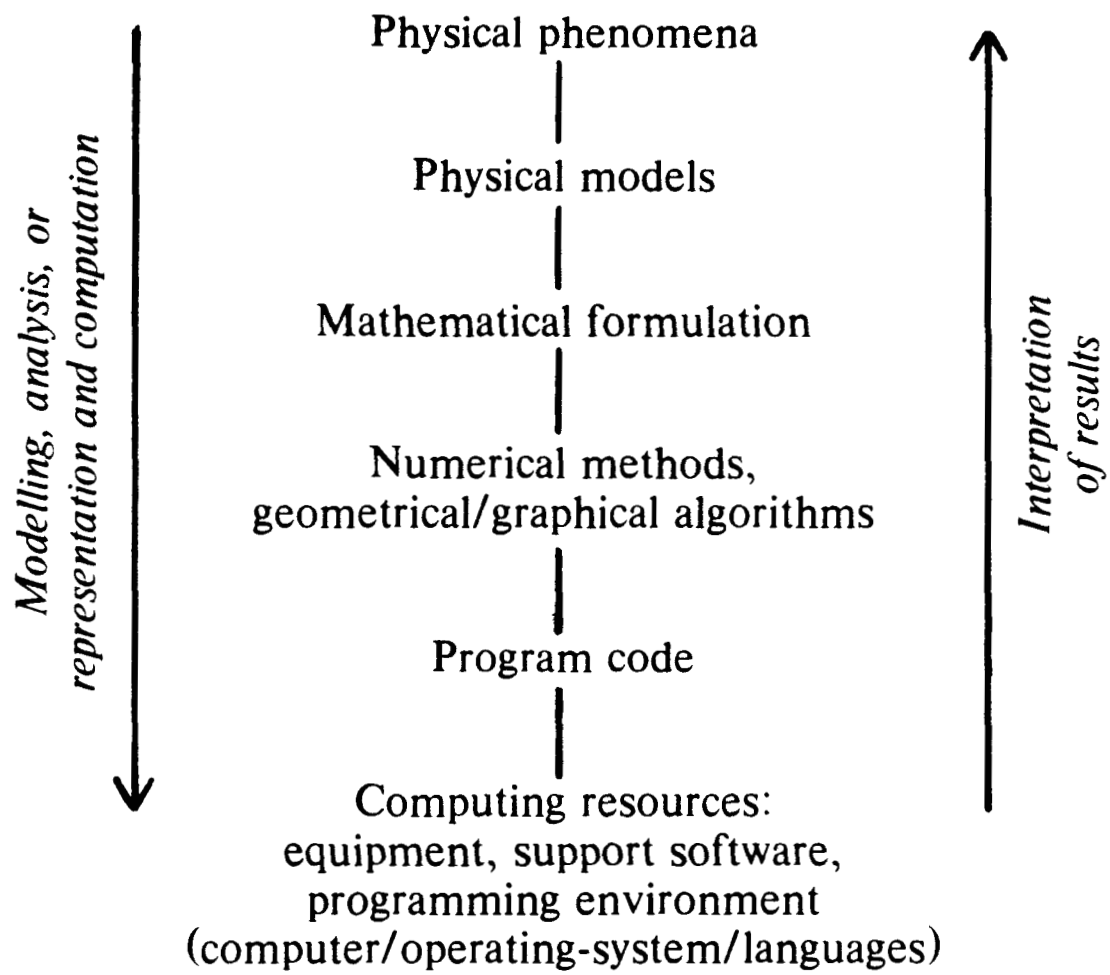


Figure 4.1 The structure of the simulation: the correspondences bridging the real phenomena to be simulated to the computing resources performing the computations

§ 4.3 Model of the System

For the system to be represented for computation and for the results to be interpreted across the layers of Figure 4.1 it is useful to have a model of the system that remains conceptually similar through all the transformations and mappings between them. The *state-variable model* (Figure 4.2) provides such a common view. The physical state of the system, its mathematical specification, and its storage representation in the program or computer system is transformed by operations performed on it, whether in the actual processing step, in a mathematical form, or as computations. In all these levels the system can be considered as producing an output and changing its state depending on the input. From an operational viewpoint, especially for program construction, it is convenient to look at the input as having two parts: one part that changes the parameters of the system and the other, an activation signal, that causes the process to start modifying the system state ultimately resulting in the next state for the system. This way the data and the procedures in the program can be matched with the state and the process respectively.

With this state-variable model for the system the interaction between a user and the system can be viewed as the attempts by the user to control the system-state through its input and to understand it by observing its outputs. The extent of such control and the ease of observation of its state may be termed the *controllability* and the *observability* of the system, respectively. A simulated system should allow the user at least as much controllability and observability as is possible with the actual process. The program, its input language and its output should be designed to facilitate such interaction with the state of the system as stored in the data-structure of the program and the process operations as coded in its procedures.

§ 4.4 Program Design for the Simulation Steps

The layered structure of the programs shown in Figure 4.1 and the state-variable representation of the system given in Figure 4.2 are two central models for the processing step simulation programs. They provide the foundation and guidelines for the semantic design of user interaction with the programs and for their construction.

A first use of the layered structure is in *classifying the users* of the programs based on their needs and depth of interest in the programs. A user who is interested only in the physical phenomena looks only at the top layer of Figure 4.1. An example of such a user is the user hypothesized in Figure 2.1. Such a user is interested only in the results of the physical phenomena and not in "any of the details" of how the simulation is done (if it is a simulation). This is a "*pure user*" of the program in the traditional sense. A scientist or research worker studying these processes in more detail and hence interested in their models, their mathematical formulations and perhaps even the numerical methods used in dealing with the resulting

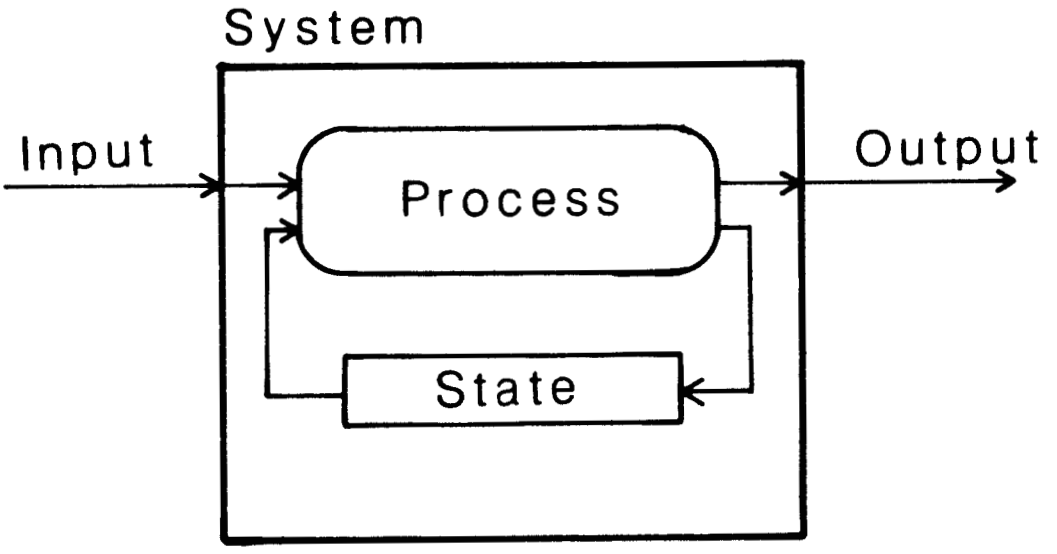


Figure 4.2 The state-variable model of the system

computational problems takes more interest in correspondingly more layers below the top one in that figure. This user's interaction may involve choosing a different model or method for solving the problem if the program provides such a choice, or specifying or building it if doesn't, or merely observing how well the methods are doing their tasks. Finally, a user, the user-analyst-programmer, whose interests reach all the way to the program code, its construction, improvement, maintenance, or even porting and installation on local computing resources will want to know and interact with more of the details of the program. In practice, there may be no single person performing the role of the user-analyst-programmer taking detailed interest in all the levels and these tasks would usually be divided between different members as engineer/physicist, mathematician, numerical analyst, or programmer as suited.

The layered structure along with this classification of users helps the program designer structure the user-interaction to be provided by the programs. In its simplest form, user-interaction is delimited by the input accepted by the program and the output generated by it. If the program has the ability to handle a choice (regarding parameter values or the type of method to be used or any other options) in its code then the user should be able to specify the choice in the input. Otherwise the only way to use that ability of the program would be to modify the portion of the program where that choice has been "hardwired" in it and then to recompile and execute it i.e. using the compiler for input — not a good way of interacting with the program (or the user). Similarly, the output should provide as much information as possible at any level, if it is available within the program, as desired by the user. Otherwise once again the user will have to resort to code modification to get (or suppress the output of) that information from the program. These simple considerations guide the programmer in the selection and the organization of the contents of the program's input and output, how it will interact with the different levels of users and also in the organization of its documentation.

The organization of the program is influenced by how the simulation of the simulation task is viewed at each of these levels. The conceptual division of the task at any level gives rise to a corresponding division in the overall programming (problem solving) effort. While the physicist, mathematician, numerical/geometrical analyst, or programmer naturally separate many of the tasks according to their fields, the situation becomes involved and interesting when some task cannot be handled well at one level but by a different approach at another level a good practical solution could be obtained.

One example of the use of a combined perspective on this interaction between levels is to note the similarity of the computational organization of the wafer exposure step for all four lithographies handled by the SAMPLE software, even though the physical phenomena seem to be of two very different types and the computational methods are very different. This similarity is useful for didactic purposes when explaining the operational features to the users, and for

keeping the structure of the four lithography simulation programs similar to each other, so simplifying their construction (X-ray and ion-beam lithography extensions have been added relatively recently as compared to the optical/uv and e-beam lithography routines), as well as in managing later additions and in maintenance.

In optical lithography, an image of a mask is formed on (in) the resist layer on the wafer, and the incident radiation causes chemical change in the photosensitive resist layer. Even assuming that the phenomenon is two dimensional (translational symmetry along the third dimension) the 2D distribution of light in the resist layer with the reflections at the various layers' surfaces and the time varying absorption in the resist are difficult to compute. The chemical effect of this spatially nonuniform and time-varying light distribution on the resist is modelled by the spatial and temporal variation of the relative concentration, M, of the light inhibiting species in the resist. The net effect of the exposure step is modelled by the final distribution of this M parameter in the resist layer (the chemical state of the resist). To make this problem computationally tractable an approximation is made in its formulation: the light enters the resist surface at normal incidence and all the wafer layers have their surfaces plane and parallel to each other (during this exposure step). With this assumption the original problem can be subdivided into two parts: (1) the computation of the incident light distribution (image) at the top surface of the resist where it is unaffected by the time varying absorption of the resist, and (2) the effect of this image at points vertically below it in the resist. The computation for the second part can be divided into two subparts: (2a) generation of a table of M distributions in a vertical "column" for various standard cumulative dose values of the light incident on the top surface of the resist (so taking care of the temporal variation), and (2b) the calculation of the actual M distributions in the resist layer by combining the values from part (2a) with the image values from part (1). This is the division of computation into the "image" machine, the "standard bleaching (exposure)" machine, and the "actual bleaching (exposure)" machine in the first few SAMPLE versions [Nand78] (Figure 4.3) (See also § 6.2.5).

For X-ray lithography the computations of the effect on the resist are simpler (no standing waves) but still divisible in similar three parts. For e-beam lithography [Rose81] the similarity is not so readily apparent. For this lithography the effect of the exposure step is modelled by the distribution of absorbed energy in the resist (for representing the exposed state of the resist). There is no physical mask and a beam of electrons "writes" the desired pattern on the resist by exposing it in spots. However, for computational purposes the spots can be considered together as the "surface image" on the resist; the Monte-Carlo computations [Rose81] that give the effect of an idealized delta-function spot of a given energy on the wafer as the "standard exposure" part of the computation; and the combination of the results of these two by a convolution operation being the "actual exposure" of the computation. The ion-beam lithography

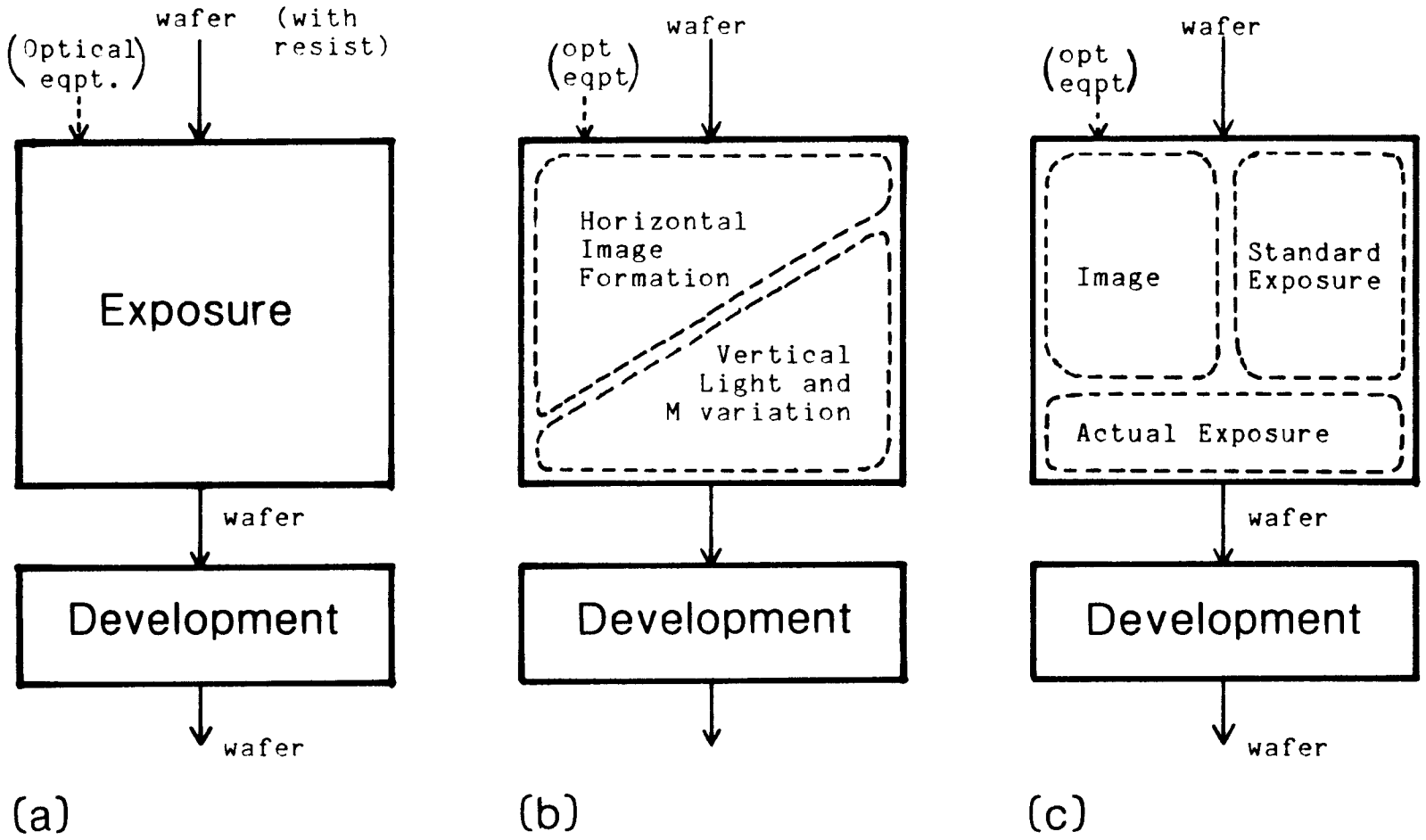


Figure 4.3 Optical Lithography Simulation
a) Physical steps (Fig. 3.5)
b) Division of computations
c) Further subdivision of the computations

simulation capability added recently uses computations similar to the e-beam routines but instead of Monte-Carlo simulation for the "standard exposure" analytic expressions are being used.

The above example where the actual physical phenomenon of resist exposure is separated into several computational tasks (image, "standard exposure", and "actual exposure") shows a few of the points regarding the interaction between the levels of Figure 4.1:

(a) The pure-user's view of the program's capabilities should be *complete* without resorting to the details of the lower levels of Figure 4.1. In general at any level of Fig 4.1 the program's simulation capabilities should be completely describable at that level (*completeness of the simulation model at every level*). This description is usually the documentation (*User Guide*) of the program that explains its intended functions (capabilities) and defines the semantics of its input and output — their organization, contents, and form. The pure-user should be able to tell the program simply to perform the whole exposure step without having to specify the individual components (image, standard exposure etc.) which are not a part of the top level. (Otherwise the program will fail to satisfy the substitution view of § 2.1). In the early versions of the program (June 1978) the user had to specify these three steps separately. *

Later the standard-exposure and actual-exposure steps were combined into one "expose" machine ** but to this day (Summer 1984) the image and "expose" steps have to be specified separately to achieve the effect of the physical exposure step. It also makes the organization of the radiation (e.g. optical) and the particle beam (e.g. e-beam) lithography systems more different than it really should be. (A definite spot for cleanup of the program and its input design.)

Another example of a similar deficiency in keeping levels separate is found in the SAMPLE documentation in the explanation of the input statement that causes the simulation of the e-beam exposure of the wafer. Instead of explaining its action as the simulation of the exposure step it merely tells that the statement causes the program to "... run the convolution ..." and it needs "the Monte-Carlo data file" ([SUG83] part 2). An explanation whose incompleteness often causes a new user a lot of unnecessary confusion and later grief due to incorrect simulation results because of incorrect usage of the Monte-Carlo data files. ***

---

* Saying that specifying the three steps in the input can be considered to be the same as specifying the physically wholesome one step is merely claiming the non-existence of the symptoms, when the real disease is the flawed design of its input language. The above is a simple example of thinking at a lower level and losing sight of the forest for the trees. The effect of such flaws is cumulative and one of their results is a loss of flexibility and capacity to extend to different situations because the design is unnecessarily bogged down in lower level details.

** Thanks to Dr. Mike O'Toole for a dinner invitation and discussion that led to this action. (Summer 1979)

*** Then why isn't the documentation corrected? Answer: for bureaucratic reasons. The purpose of pointing out the above example is merely to show the clarification afforded by the perspective offered by Figure 4.1 even in

(b) The same layered structure can be used to give a meaningful structure to the output produced by the program. The output for different steps in the processing sequence is naturally concatenated in the same *sequence*. However, even when there are multiple computational steps in the simulation of a simple processing step (e.g. for the lithographic exposure process as described above) the outputs of these individual computational substeps should not be mixed in a haphazard manner. Usually such mixup arises when the program is modified without attention to this simple output-structuring guideline (concept). The mixing unnecessarily spreads related pieces of information textually apart in the output and results in a loss of cohesiveness in the presentation of the information. A current example of this is the output of the e-beam lithography exposure simulation machine's output in SAMPLE versions upto 1.5b. This can be corrected easily by rearranging it (and is (slowly) being done now).

Another element of structure in the textual output produced by the program is the *nesting of the output information according to the top-to-bottom hierarchy of layers in Figure 4.1.* The output from one level provides the context for the output from the level below it. (Except that at the lowermost level some code installation and implementation problems may show up to be more related to the computing resources being used rather than the simulation being performed. In any case, the output should show this nesting according to its context.) Indicating the nested nature of the (textual) output by indentation (similar to the indentation used by programmers for program code in a block-structured language where each nested block is indented one level further than its surrounding block) makes the output easier to grasp visually. (*The importance of visual layout.*) For example, in the output for a simulation step the parameters of the physical process being simulated should be at the outermost nesting (hence indentation) level. Then as the information at the lower levels is being output it should be nested inside that. The details of the numerical method i.e. discretization, grid-size, the location of grid points, the number of grid divisions should have a further nesting (indentation) level, and details of program code (e.g. the number of array elements used for storage, diagnostic messages like "subr xyz called with parameter values ...''), when output, should be indented even further.

Usually due to the limited width of the textual output device some compromise in such indentation is necessary. Also for the visual balance in the *output layout* some information (e.g. plots) may be better presented centered within the available output width of the display or printing device. Even with such compromises in the output layout due to the nature of the

---

documentation, and how a deviation can be harmful. Even though the project administration is an integral part of a technical project that is not the topic of discussion here.

output medium and for reasons of visual aesthetics, the concepts of sequence and nesting enhance the output understandability for the user. And in the process of attempting this coherent output structure the insight gained in the computational organization and hierarchy is in itself valuable to the programmer/designer.

(c) An interesting situation occurs when the program exhibits some peculiar behaviour that seems like an error at one level but may simply be an "interface quirk" between the levels, or just a limitation at one level being propagated to another level where its peculiarity is enhanced and becomes noticeable or bothersome.

An example of this is a problem noticed by Dr. Douglas A. Bernard * in Fall 1983 when trying to see how the photolithography simulation was performing at the vertical boundaries of the computation window. He simulated two full periods of a simple mask being printed on a resist layer (Figure 4.4a) so that the region at the center had the same physical processing conditions as the region at the two boundaries. Any computational errors at the boundaries could be detected easily by comparing the boundary portion of the profiles with the center portion. The horizontal image from the mask (sketched in Figure 4.4b) as plotted on the screen of a high resolution digital graphics display didn't show any problems at the boundaries. But in the developed profiles the center portion lagged significantly behind the boundary regions so that the developer broke through the resist layer (in the simulation) at very different times at the boundaries as compared to the central region (Figure 4.4c).

This was very confusing because the profile advancement algorithm used for the simulation of the development tries to move the boundary points vertically downwards. So the computations should not be affected by the boundary location of the points when they are moving vertically downwards in the actual processing. Therefore these points should have the same movement as in the center region in Figure 4.4c.

The explanation of this discrepancy between the actual simulation and its expected behaviour lies in the discretization used for the location of the grid points at which the horizontal image intensity is computed by the program. The expressions used for the evaluation of the image are indeed symmetric and periodic along the horizontal direction. The grid points are uniformly spaced in the window. But because the program uses an even number (50) of grid points, hence an odd number (50-1=49) of grid divisions, in the horizontal window with the end points located on the two boundaries there is no grid point located at the center of the window. Because the image intensity is concave downwards in the central region the intensity at the two grid points on either side of the center is slightly smaller than the intensity at the peak

---

* of Philips Research Laboratories Sunnyvale, Signetics Corporation, California, U.S.A.

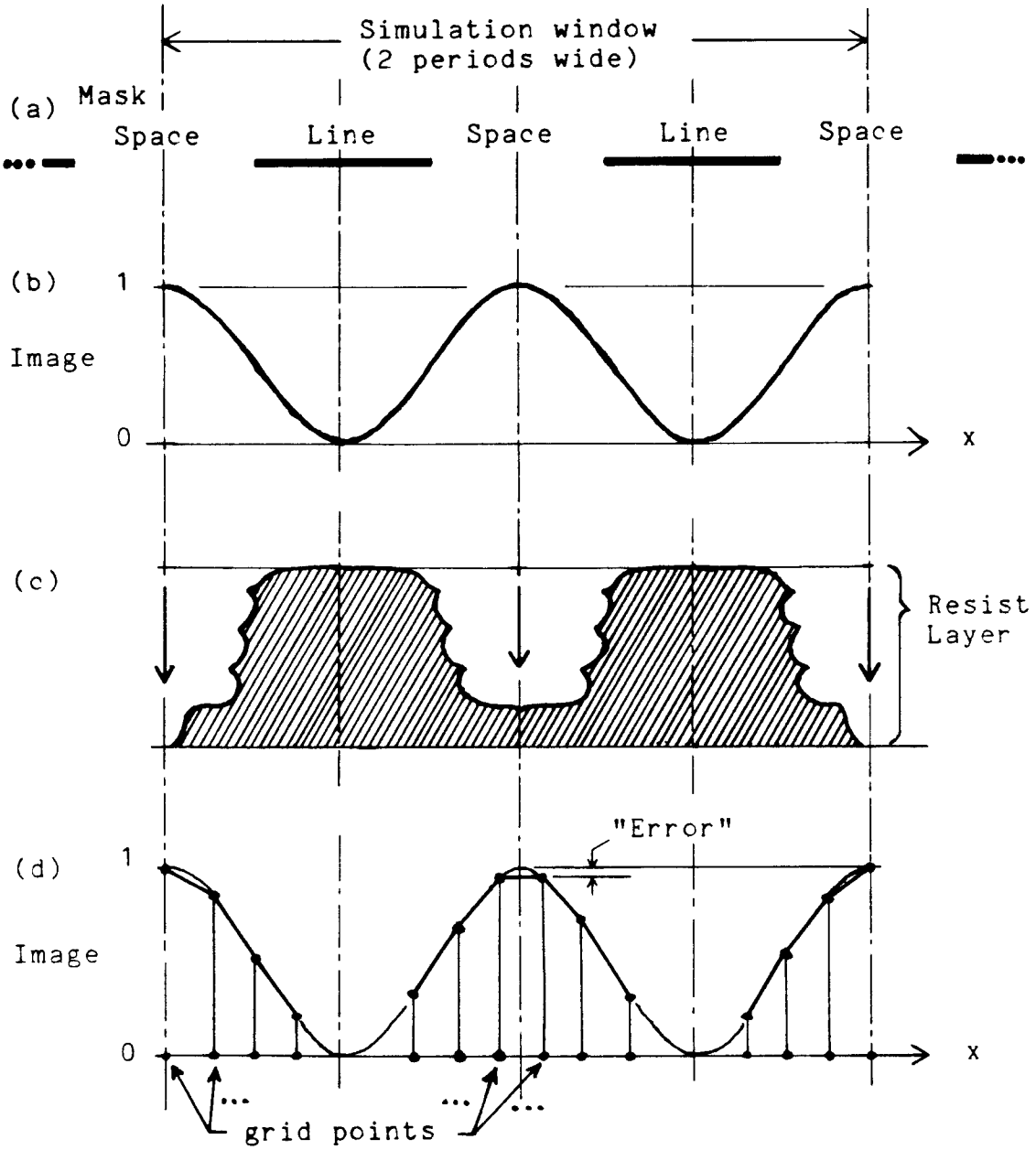


Figure 4.4 Example of an unexpected interaction between the levels of Fig. 4.1. Sketches of the:

a) Cross section of Mask. Simulation window.
b) Image from the mask
c) The simulated resist profile
d) The source of error (due to the location of the grid points, An exaggerated sketch)

(which is at the center). But the intensity at the boundary points has the full peak intensity value. Though this is a very small (second order) difference and is easily missed in a visual inspection of the image plot, it is propagated to the computation of the vertical standing waves of intensity and M in the resist. Then in the development process where an extremely non-linear relation between M and the development rate amplifies this difference enough so that the cumulative development time for breaking through the intensity (and M) node locations along a vertical path happens to show up loudly at the last node. Once this was realized the flat top in the center peak of the image plot of Figure 4.4b was easily noticed (sketched separately in Figure 4.4d).

This showed that even though the numerical (and the simulation) performance of the image and development routines is individually very good, combined together they did not stand up well to the critical test of this simulation. And the explanation was to be found in an unexpected place. ** Looking back at how the explanation was arrived at shows a clear help from the layered structure of this simulation shown in Figure 4.1. The symmetry of the physical situation was unviolated at the physical/mathematical formulation level, and also at the geometrical/numerical computation method, but not in the choice of the numerical parameter values chosen in the discretization (grid point locations). So without wasting time at the lower levels for "code debugging" the explanation could be found quickly. *** Moreover this points out that an input statement allowing the user to choose the locations (or number) of grid points would be a worthwhile addition to the program for improving the controllability of the discretization.

§ 4.5 Rumination

An analysis of the problem to be solved will reveal many of its characteristics that would lead to a solution. Many characteristics of a solution found, if any, would get shaped by the approach. The various aspects of the problem and the solution uncovered by the analysis provide an intrinsic framework for organizing further efforts. In the preceding chapter (Chap. 3) a framework was shown for the process sequences and the communication of simulation wafer results from one simulation machine to another. In this chapter the structure of the simulation efforts for the individual processes was examined. *The various layers or levels of the simulation* (Figure 4.1) and *the interplay of representation and interpretation between them* has influenced the nature of most efforts in our group in studying the physical phenomena involved in IC

** This can be a very touchy situation in a multiperson project.

*** For lack of time a rigorous verification has not been carried through. Without it this explanation remains only a hypothesis. A familiarity with the values encountered in such simulations tends to make me believe in the correctness of this hypothesis.

processing.* The concepts outlined here, the static model of Figure 4.1 and the execution plans in Figures 3.4 and 4.2, have provided the framework for the software written (and being written) and have helped in classifying various new ideas and approaches from many people by providing a clearcut perspective to evaluate their potential contribution to the overall effort.

In the next chapter we examine the other part of Figure 4.1 — the computing resources, by focussing on the software issues and the software project aspects of this effort. □

---

* See the references listed at the end of part 1 of *SAMPLE User Guide* [SUG83]. Also, other research workers using computations to model physical phenomena seem to converge to a similar philosophy [Xxxx82].

Chapter 5

Structure and Implementation of the Software

§ 5.1 General

In discussing the basic design and structure of the simulation efforts and the program, the contents of the user interaction were considered mainly from the simulation goals point of view. In this chapter the user interaction will be considered from the computing resources point of view. Fortunately, these two views give mostly an orthogonal set of components for the program structure for user interaction, and hence for the user interface. So it is possible to choose the form of interaction from a variety of styles and still satisfy the same information content requirements for the user.

The software aspects of the structure and implementation of such interactions, the management of the software, its evolution, and its design for growth are the topics of discussion in this chapter. Details of the code and other aspects that can be seen easily by studying the code, or various standard textbooks, are avoided. The intention is to convey the general philosophy and directions followed, so that the program framework can be understood easily and a perspective obtained for the continuation of this software project.

§ 5.2 I/O: Devices and Device Handlers

The primary interaction device that we* have been using for communicating with the computer and the program(s) is a video terminal with an alphanumeric keyboard for input to the system and a display for output from it. The communication is by *sequences of characters* sent over a communication line. A terminal, along with the ability to use disk files for feeding input to the program and for storing the output from it, a line-printer for getting hardcopies, and postprocessor plotting programs for displaying graphical output on the terminal screen help achieve an adequately comfortable interaction with the program at present (Fig. 2.3).

---

* The SAMPLE Group at the University of California, Berkeley, California, U.S.A.

Even though we do not make use of various new peripheral devices like graphics tablets (digitizers), light pens, or electronic mice, it is interesting to consider how they could be used if they become sufficiently accessible and available in the future.

Graphics tablets or lightpens could be used to enter various graphical and geometrical entities to the program. For example, the initial profiles for etching or deposition, or the development rate as a function of the M parameter could be specified by entering a curve graphically rather than by describing it with an alphanumeric (textual) input. Since such devices communicate their information as character sequences (say, as encoded coordinate values in their frames of reference) in theory they do not change the nature or meaning of the input quantities for the program.

The "I/O devices" for interaction need not be physical devices. They could be general computing resources like a screen management software package (e.g. the *curses* library [Arno81] on 4.2BSD Unix), that makes screen oriented menu-style interaction easy to implement. Again, they do not change the nature or meaning of the information, though they can make a vast difference for the convenience of using the program(s).

Interrupts are a different type of mechanisms for communicating with programs due to the *asynchronous* nature of their interaction. They could be used to signal the program to change its course of computations interactively. Consider the following scenario: The program displays the resist or wafer profiles as it computes them with small steps of the simulated time (i.e. it has *run-time graphics* capabilities), and the user observing these profiles suddenly decides to stop that simulation at a particular profile shape to investigate the effect of the next simulated process step (e.g. a deposition step following etching) on that shape. To do this without interrupt mechanisms would involve either (a) trial-and-error for the choice of total profile advance time (simulated time) on the user's part if all intermediate profiles are not shown; or (b) a voluminous output from which the user decides on the desired simulation time; or (c) asking the user at every simulation time increment step whether to continue with the profile advance step or to go to the next processing simulation step on the program's part*. None of these other methods seems to be as direct, elegant, and convenient as using interrupts for this type of inherently asynchronous interaction**. In the absence of general purpose device support libraries or other

---

* Other possibilities to imitate the asynchronous interaction may be: (a) at each profile advance step the program would wait for user input (a "blocked"-read) and if there is no input for a certain time a *timer* process would *wake* it up to abandon the read and continue the computations; or (b) the program would wait a certain small amount of real time (to allow the relatively slow response speed of the user) after each profile step and then try to see if there is any new input typed by the user (a "non-blocked read"). If there is none then continue on the profile advance computations.

Prof. Oldham :- "This corresponds to a real laboratory process, viz. *end-point detection* in manual mode. The considerations above have their counterparts in the real laboratory. The equipment dependence makes it difficult there too."

** Indeed, many of the special user-interaction techniques used in computer- and video-games could be adapted for

language facilities (say, for interrupt handling or for graphics output) the programming difficulty or lack of portability arising from overdependence on a particular system configuration would tend to deter the programmer from utilizing many features of the available devices. However, being aware of such possibilities would help in writing software that could be conveniently extended in these directions at some future time.

§ 5.3 I/O: Communication Channels

The communication between a program in execution and the user, using the physical devices, occurs along fairly similar channels on most contemporary computer systems *. Each computer system provides an environment for the user with system-specific means of invoking the program, storing data in its file system, and other means of storing and communicating data between the user and the program (Figure 5.1).

Other than the brute force approach of having the data "hardwired" in the source code of the program, the more flexible means of getting it to the program are by using an input device like a keyboard, or disk files, or command-line arguments to the program. Similarly the output from the program may be channeled onto the standard display output device or various files for different types of output information. The files may be physical devices as is easily allowed in the Unix system. Figure 5.2 shows a generalized view of these I/O channels including the asynchronous signals (interrupts) from the user of the program, from the system, or from the processor (e.g. a floating point overflow trap), and possible invocations of other programs from the user-invoked program, say, to send automatic mail to the program maintainer in case some abnormal conditions are detected during the program execution.

The outputs for "direct human consumption" and for "post-processing" by other programs are often not mutually exclusive. Further, different types of information to the user may be grouped together and directed to a single channel (device or file) for implementation convenience. Such *folding* (or multiplexing) of different channels onto one may arise when the output relates to different levels shown in Fig. 4.1 or when the program is producing different types of information in alternate steps. Without proper output design and organization this would result in a confusing mixup of the information on that physical device (or file). For example, if the echo of the input lines from the standard input and the computational results from the simulation are both sent to the standard output then if the input interface routines perform a lookahead on the standard input that lookahead may result in the echo of input lines

interaction with the simulation programs.

* For the purpose of this discussion, the term *channel* is used for denoting ways of getting data to and from the program when using it, free of the operational details of the I/O devices but still dependent on the computer and system support.

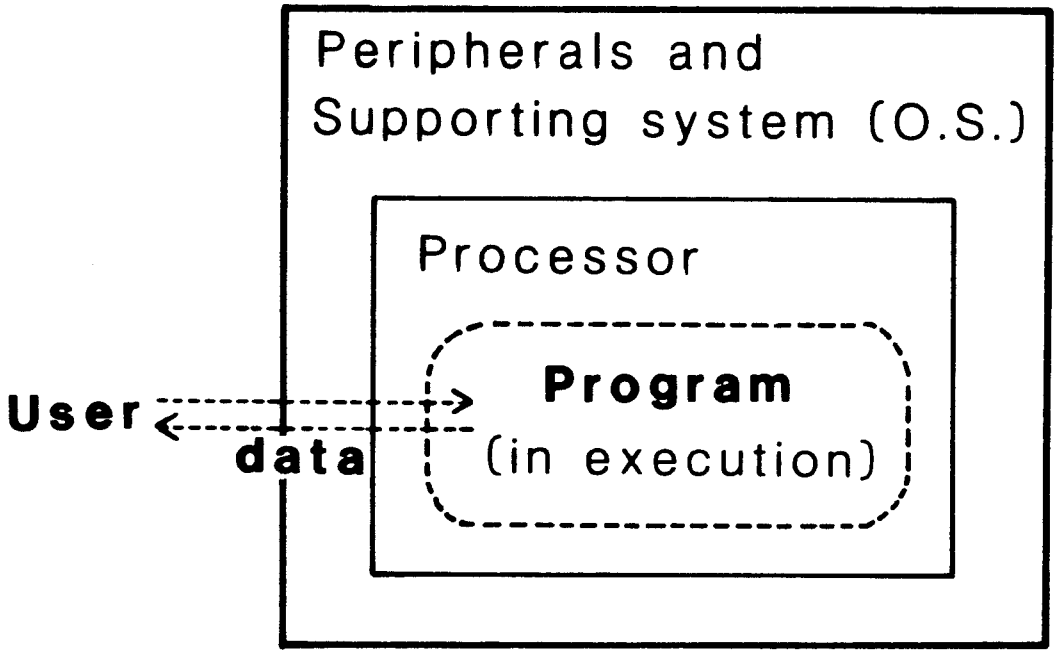


Figure 5.1 Intermediaries for data communication between the user and the program

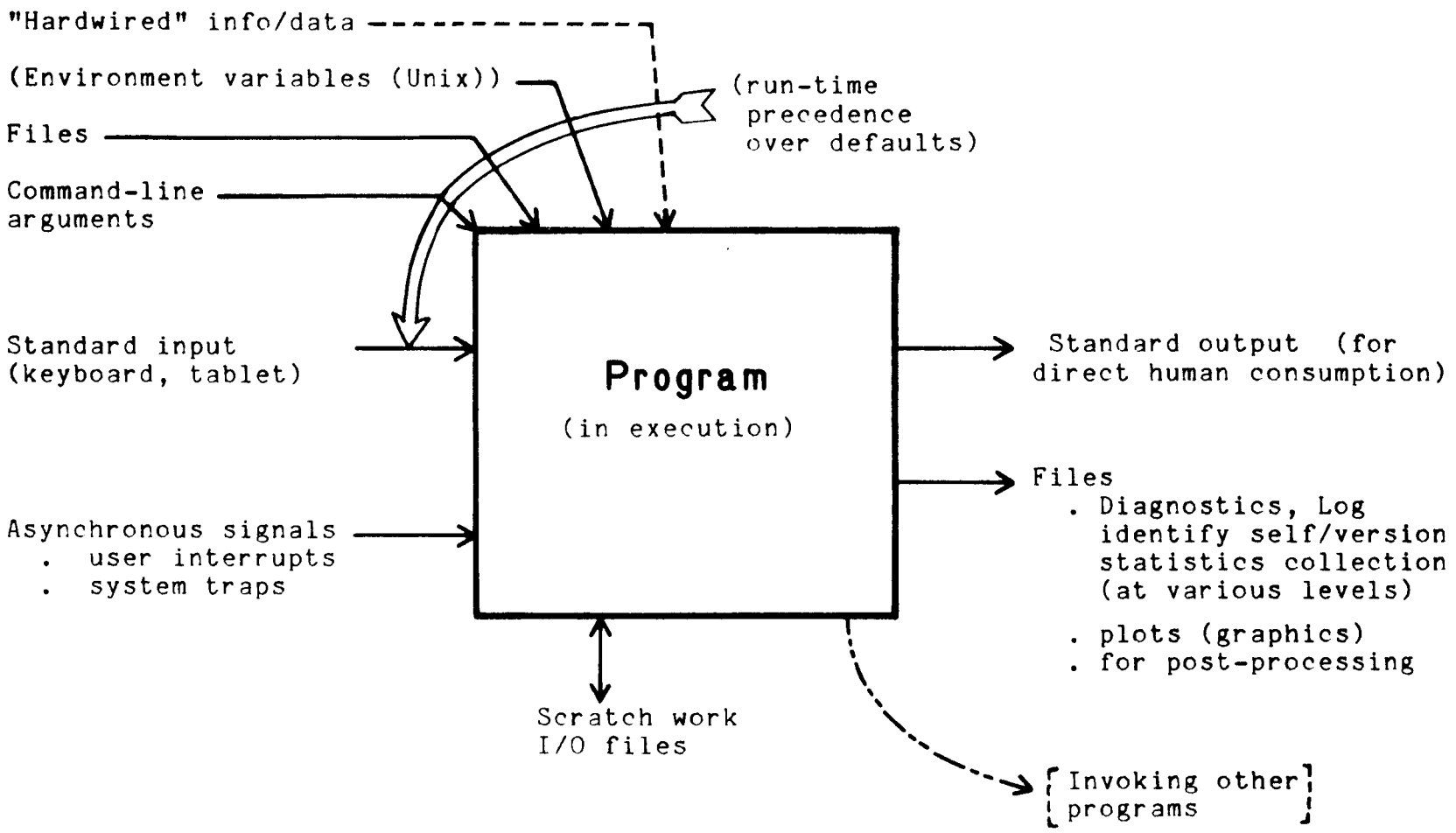


Figure 5.2 I/O comunication channels for a program

being out of step ("ahead in phase") with the simulation output. The situation could become more confusing if there are diagnostic messages (for the input) interspersed in the same output channel and they are not at their right place because of the missed synchronization between the echo of input lines and the simulation output. One solution, currently followed for SAMPLE input statements, is to extend the input language syntax by defining an explicit token for separating (or terminating) the individual statements which would indicate the end of a statement to the input routines, thus obviating the lookahead*.

An example of the mixing of the outputs from the same level occurs in the simulation of the optical image for a multi-wavelength case when the user wants data for plotting both the Optical Transfer Function (OTF) and the image component at each wavelength. Since the OTF and image component are both computed for one wavelength before proceeding to the next, outputting them to an output channel when they are computed will mix them up with each other (e.g. if they are to be sent to the same output channel, like the card-punch in Figure 2.2b). A solution is to store (at least one of) them in a buffer space till the output for one is finished and then output the buffered data. Another solution would be to output only one of them (the image) during the computations and when that output is finished recompute the other (the OTF) and output it then**.

Similar problems arise in handling the input to the program when it gets voluminous. A main input channel to the program is the standard input via which the user specifies various parameters and gives simulation commands to the program. The syntax and semantics of this language has to be designed for convenience of usage as well as precision of communication. Just like the mixups of the outputs from various levels there could be mixups in the intended level (in Fig. 4.1) for an input command. A "produce verbose diagnostic output" command will have to be made more precise as to the level at which it is intended. The diagnostic extra output may be intended for debugging or checking the input-interface only, or the detailed physical results from the simulation. And a specification of a "mask" or wafer parameters in optical lithography would have to be distinguished from similar specifications in x-ray lithography. The point here is that folding or multiplexing of various types of inputs may occur onto one input channel. This input will have to be "demultiplexed" by the program's input interface (or in general user-interface) to send the information to the appropriate parts. In the program

---

* Whether it is a statement-separator token (and optional) or statement-terminator token (and optional) has no practical effect on its usage.

** Digression:- Actually, the way the program developed over time, a doubly redundant combination of buffering and recomputation is used for the plot data channel. The image components are buffered, and the OTF components are recalculated and buffered too. These vestigial features of the code organization arose due to uncertainties in the direction of growth for the program in the beginning, and make those parts unnecessarily more involved. They should be cleaned up now. /End of digression.

design this may be "solved" by an elaborate input language design (e.g. by having distinct statements for each distinct elementary input type, as is done now, or by having some kind of context hierarchy with some kind of block structure to allow simple elementary statements to be applicable to different parts of the program depending on their input-context). Or another solution would be to separate the inputs to different channels (different files are also different channels) for different levels, say, by having all input interface debugging options read from a certain file, or allowing the wafer profiles to come from a specified file while other commands are entered on the "standard input". This illustrates how simple considerations like the number and types of input channels to be used will quickly affect aspects of the input language design.

When there are many potentially parallel input channels to the program for specifying the same item, a decision has to be made as to which specification will supersede which other specification. Usually this is a simple matter of using the input from the more flexible channel to supersede the input from a less easily changed channel. For example, the "hardwired" default values in the program's source code are superseded by their respecification using the command line arguments (if the program can handle them), which in their turn are superseded by the input entered via the standard input channel as the program is executing. Further, if the output results of the program are to be reproduced in a different run for documentation, demonstration, or verification, all the relevant inputs on all the input channels will have to be reproduced as of the reference run. Though obvious in principle, due attention is needed in practice to avoid errors, especially when comparing with a slightly modified version of the program or outputs from two different installations.

Currently (Summer 1984) no asynchronous signals are used by SAMPLE, neither are any command line arguments (or the Environment variables of the Unix *shell*). All disk-files used for I/O are simple sequential in nature and in human readable text format to simplify the development and verification of the program and for communication and transmission to different installations. All the input channels, except for the interrupts, are character sequential in nature and so, in principle, the same language defined for the standard input could be used for communication over all of them. However, for efficiency or coding convenience it is sometimes more suitable to allow different syntax on some channels. For example, the e-beam delta-function exposure, the data obtained by Monte Carlo methods, is more conveniently looked at as an "array of numbers to be read" rather than an "input statement to be interpreted".

The association ("binding") of input channels to physical devices may be under program control. This is particularly true for disk files used by the program. The program may use fixed names for the files when looking for specific input information (e.g. e-beam delta function

exposure data) or when producing specific output information (e.g. for the plot data). This *static* binding is useful enough for separating different types of information, but it is far more convenient for the user to be able to specify the name bindings in the input. Such *dynamic* binding may be used, say, to change some wafer parameters during an e-beam lithography simulation run and then use a different delta function exposure data file suitable for the new wafer parameters, or to tell the program to output the plot data for the optical image in a file separate from that used to store the OTF curves or the wafer profiles. Another example of such *meta-input* capabilities to "specify an input channel in the input" could be the usage of "include" files to specify another file with (previously stored) input commands to be executed as if those commands came from the original input channel itself (See [Kern76] section 3.3, pp. 74ff). These meta-input capabilities do not change the simulation functionality but they do enhance the user-interaction power of the software.

§ 5.4 Styles of Interaction

The interaction style of a program is closely related to its functional design. The operational design, which enables the user to make use of its functional capabilities reflects the view of the system as built in the program. For the SAMPLE program(s) the user-interaction is designed with simple atomic transactions (*unit-transactions*) of either parameter setting or activation signals being given to the program to match the state-variable model (§ 4.3) for the system.

Accordingly, the input language (for the standard input channel) is a simple non-procedural command language where each statement either sets some simulation, system, or program parameter(s), or activates a state transformation corresponding to a physical processing step for the system being simulated (these are the statements that tell the program to "run" a simulated "machine")***. Since the state-variable model can be applied at all levels of Figure 4.1 this can give as much controllability and observability over the program as desired****.

The operational part of the user-interaction may be considered as merely interacting with the computing resources — the peripherals, file system, and the processor. The input-interface could be a simple command line-interpreter that waits for a new (textual) command statement from the input channel, or a fancier menu-oriented front-end for the program (or even a

---

*** The minor deviations where the input statement causes some parameters to be set and then a machine to be "run" (e.g. the statement causing the thermal diffusion machine to be run for optical lithography also specifies the diffusion sigma) are not significant here. But for the inertia/reluctance involved in changing a working and documented program, they can easily be split into parameter-setting and action statements.

**** The program could use some more statements for merely enquiring the current values of the state. Their absence is not strongly felt yet because when an action statement is executed the values of most state-variables of interest are printed out. Another reason being the tendency to use the program in batch mode with already prepared input files for the standard input.

multiple-window style interface for providing concurrent channels for inputs at various levels of Figure 4.1, or for various machines of Figure 3.4, through different windows if enough hardware and software support is available for writing it). The difference between the simple text-statement-at-a-time and the other (menu or screen oriented) types of interaction styles arises mainly from who takes the initiative for the interaction — the user or the program. For the command interpreter style the user has to (remember and) enter the input statements whereas in a menu the desired action has merely to be chosen from an enumeration of allowed actions (with a few numeric parameters specified by other means). While convenient for novice and casual users, the *forced* interaction of menu-style interfaces is a hindrance to the expert user who may want to enter input before the program prompts for it, and totally unsuitable for running the program in a batch mode, say, for running a suite of test or demonstration input files overnight without explicit user intervention. These considerations along with the simplicity of programming required and the minimal demands made on the input peripherals have resulted in the choice of a textual command-statement input interface for the SAMPLE program(s).

§ 5.5 The Input-interface

Once the decision is made to treat the input as a sequence of unit-transaction (textual) command statements, the input interface is easily constructed in a traditional fashion with a lexical analyzer (and a set of operating system and file-system interaction routines), parser, and semantic routines [Nand78].

Since the input statements are chosen from a finite set of parameter-setting and activation-signal action statements the input language is a type 3 language (i.e. a regular language recognizable by a finite state automaton). To simplify it even further (in version 1.5b) all statements have exactly the same syntax: a header keyword followed by some numbers; and a common internal representation in the input-interface: an "action-index" (the "TRIAL number") followed by a list of (the rest of) the parameters as they appeared in the input. This simplicity is reflected in the input by having another equivalent form for all the keywords (except "TRIAL") as the keyword TRIAL followed by the action-index as a number *.

To execute a statement, essentially the top-level controller branches to a portion of the code corresponding to the action-index, and performs the action associated with that index using the rest of the (numerical) parameters, if any, in the input statement. These actions

---

* So, in theory, even the keyword TRIAL is not necessary! (That is because the statement separator token of § 5.3 and § 5.7 is sufficient to delimit the statements.)

range from the simple, e.g. setting the optical wavelength to be used, that are performed by a few statements of program code, to the extensive e.g. the simulation of an optical development step, that have their own modules with many routines in them.

The connection between the input handling routines and the semantic routines that perform the actions indicated in the input statements is by a simple list of numbers containing the action number and the rest of the (numerical) parameters, and a controller that tries to get the next statement (i.e. the action number and parameters from the next statement) and then execute it. The controller tries to execute all statements unless an error is detected.

In any invocation of the program the top level actions are (1) the initialization of various levels — the I/O units, the input interface, the simulation lab; followed by (2) the repetitious loop of getting the next statement and executing it, as long as the end-of-input is not reached or the program hasn't stopped due to some other serious non-recoverable error in the simulation routines (see the code for other small details of handling the error-flags for the input interface); followed by (3) the termination routines that do a little clean-up like closing any open output units, or printing some execution time statistics for the user. The details of this control structure for these actions should be clear from the amply commented code module, *mod01*, for the top-level controller (see also [Nand78]).

§ 5.6 Automation to Help the Programmer

The simplicity and uniform handling of the input statements at the top level of the program's design eases the tasks of building, maintaining and extending the design and the code. It also helps in identifying the tasks where program generators could be used to automatically generate portions of the code that are well understood. A judicious use of such program generation tools, if available, could enhance the power and usability of the input interface with comparatively little effort in applying them.

Currently the only use of any automation in code generation for the program is the usage of a small preprocessor program that takes a list of keywords and their corresponding action indices (TRIAL numbers) and generates the (Fortran) data initialization statements for the tables in which the program stores them. The file* containing the user- (programmer-) specified keyword/action-number pairs looks like (only the first few lines in the file are shown here):

---

* File ucbesvax:~samsoft/release/ucb/kwd/kwd2trial

```
# Keywords to Trial numbers mapping for SAMPLE.
# Experimental version (Sep 05, 1981) (SNN).  (Jan 17, 1983.  SNN)
#
#234567890 ####
end           -2  (currently just like the stop stmt below)
stop          -1  stop the simulation, exit from the program
recover        0  'recover' from syntax error
exectimes      5  system execution times (cpu, system, real time)
lprwidth       6  line-printer width (columns) to be available for plots
ifcdbd         7  user interface debugging output depth
help           8  runtime help
#
# The previous kwd-style stmts are now treated like mapped-kwd-stmts.
lambda       201  wavelength specification
dose         202  exposure amount
proj         204  projection type optical printing system
contact      205  contact type optical printing system
line         206  a single line mask
space        207  a single space mask
linespace    208  a periodic pattern of lines and spaces on the mask
```

where lines starting with "#" are comments in the file. The Preprocessor program converts these lines into the following (Fortran) DATA statements (again, only a first few lines are shown here):

```
      block data mpkwtr
cb
c
      include 'cblexsc2'
c
c # Keywords to Trial numbers mapping for SAMPLE.
c # Experimental version (Sep 05, 1981) (SNN).  (Jan 17, 1983.  SNN)
c #
c #234567890 ####
c end          -2  (currently just like the stop stmt below)
      data jrwt2( 1, 1) /1he/
      data jrwt2( 2, 1) /1hn/
      data jrwt2( 3, 1) /1hd/
      data jrwt2( 4, 1) /1h /
      data jrwt2( 5, 1) /1h /
      data jrwt2( 6, 1) /1h /
      data jrwt2( 7, 1) /1h /
      data jrwt2( 8, 1) /1h /
      data jrwt2( 9, 1) /1h /
      data jrwt2(10, 1) /1h /
      data mpw2tr( 1) / -2/
c stop         -1  stop the simulation, exit from the program
      data jrwt2( 1, 2) /1hs/
      data jrwt2( 2, 2) /1ht/
      data jrwt2( 3, 2) /1ho/
      data jrwt2( 4, 2) /1hp/
      data jrwt2( 5, 2) /1h /
      data jrwt2( 6, 2) /1h /
      data jrwt2( 7, 2) /1h /
      data jrwt2( 8, 2) /1h /
      data jrwt2( 9, 2) /1h /
      data jrwt2(10, 2) /1h /
      data mpw2tr( 2) / -1/
```

where the array elements jrwt2(i=1 to 10, j) hold the j-th keyword, and mpw2tr(j) holds the corresponding action number. At the end of the array element initialization statements another statement is generated to initialize the variable that holds the total number of the entries (= the maximum value of j) in these tables, followed by statements for the type and size declaration of the array variables holding these table entries. Then using a file inclusion program **, and Unix utilities like *make* and *ed* the statements are put in the proper sequence for a Fortran BLOCK DATA subprogram, without programmer intervention.

For help in reconstructing the original file with the keyword/action-number pairs, the original lines (including comments) from it are output in the form of Fortran COMMENT statements. If desired, these can easily be put back in the original file form by using a Unix pipeline ***. Such "inversion" capability is very convenient for checking the mutual consistency of

---

** See [Kern76] section 3.3, pp. 74ff.

*** (grep "c ' bdfile | colrm 1 2 | tail +n)
where bdfile contains the BLOCK DATA subprogram generated by the procedure above. The value of n for "tail +n" is chosen to remove the few extraneous lines not part of the original file. That could be avoided by having a distinct character other than a space in column 2 of the comment line holding the original lines.

these files.

The above example is admittedly very simple considering the current state of the art [Aho77]. But it serves to indicate the power of automatic code-segment generation whenever it can be used to help perform a well defined and well understood task. (And the technique of allowing easy inversion of various files is helpful in maintaining and verifying these files.)

§ 5.7 Input Language Enhancements

The input language to the program with one-action per statement and a syntax stripped down to a minimal form of a keyword followed by numerical parameters only is very austere indeed. With good mnemonic choices for (some) keywords, capability for adding comments, and formatting freedom for the input statements (putting more than one statement per line or using more than one line for a statement, and a token — the semicolon, for use in explicitly terminating a statement at a given line to denote the absence of further, hopefully optional, numerical parameters) it has continued to serve well over the years. By judiciously choosing the most frequently used values as the default values for the parameters needed by the program, and a consistent and simple user-model (the state-variable model) for the inner workings of the program, most of the input sets are only a few lines (about 5 to 20) long. Because meaningful simulations can be carried out with such short inputs to the program no crucial needs arose for changing the input language.

However, many enhancements, both cosmetic and deeper, in terms of the user-interaction power, can be made to the input language without taking away its "hands-on" unit-transaction flavour for simulation.

Macro handling and file-inclusion [Kern76] would make it easier to use common input data for a set of related simulation runs. Expression parsing and evaluation by the input interface would make specifications like "quarter wavelength thick" oxide layer easier to input and keep consistent if some values are changed. Extending the expression-specification further, functions could be defined in the input for use during the simulation. For example, this would make it easier to change the development rate function in optical lithography when investigating the different rate functions. Such coupling between the input (language) and (the programming language used to write) the code would encourage experimentation with various physical models by making it easier to perform. It can be implemented by constructing an intermediate form for the function/expression in the input interface and interpreting it as often as needed during the simulation.

Some of the ideas of structured programming can be readily applied to the input language. The basic concepts of sequence (grouping), conditional execution, and iteration [Dahl72] are as applicable to the input language for this (set of) program(s) as they are for a general purpose

programming language. The input-interface may be likened to a command-interpreter for a computer operating system (e.g. the *sh*, or *csh* for Unix). For the "simulation machine/lab" provided by the program it interprets the commands given by the user. The user may want to execute a group of statements if some condition holds (e.g. the contrast figure of the optical image is in a certain range, or the occurrence of an event like the developer breaking through the resist). Or the user may want to repeat (iterate) some steps with change of some processing parameter (e.g. starting with an initial profile examine metal deposition profiles with different angles of deposition). Enabling the input-interpreter to make such user-specified decisions would let the user explore the simulations more freely. The uniform nature of handling the current single-action input statements is amenable to these structure extensions to the input language in the same way that such structures are handled by *csh* in (the Berkeley) Unix, or the way that *ratfor* extends Fortran [Kern76] except that instead of translating the structured (compound) statements into their elementary statement components and outputting them, the elementary statements are executed (interpreted).

§ 5.8 Software Modules

The top-down decomposition of the programming task gives rise to some of the natural modular divisions according to the functions to be performed. The division into a user interface and a simulated laboratory is the main modular decomposition for the program. The user interface has its own sub-modules of lexical analyzer, parser, and related routines. The simulated laboratory is a collection of simulation modules as simulated machines.

This much top-down division gives a good organization for the program to implement a few initial versions and get a better understanding of the programming task. As the program evolves many common programming tasks are noticed within similar modules (e.g. profile advance routines for most of the profile shaping processes). For the current version of the SAMPLE program such common or related tasks are very noticeable within each level of Figure 4.1. Grouping the routines handling such generally useful tasks into their own modules has obvious benefits for the software development and management. These modules represent the abstractions of the tasks performed by them in a conveniently usable form.

Currently such "bottom-up" modules in SAMPLE are a few collections of routines ("libraries") in their own source code files. They are:

- math library / numerical routines (mod22)
- geometrical/graphical library routines (mod23)
- output and plot library for simple functions e.g. for producing line-printer plots with character arrays, putting the profiles in plot-data file (mod21)

- system specific routines e.g. date/time routines, flushing output buffers and some character string handling routines (mod08)
- buffer subroutines for matching residual discrepancies between the user interface and the simulation routines as they tend to evolve independently (mod07).

Once the program is working the efforts to "clean it up" by grouping such general routines in general purpose modules and libraries smooths the way for further growth, code improvements and maintenance. It signifies the maturing of the software that may not be apparent to the "pure user", because the (simulation) functional capabilities of the program may not show the change immediately.

These modules (have the potential to) become independent software products in their own right. They allow the higher levels of the software to be reorganized easily. This could give rise to some powerful combinations of the software for the end user. An example of this is the high resolution plotting software for displaying curves and profiles on a graphics terminal screen or a digital plotter. Initially separate from the simulation program because of the configuration of the simulation computer and the graphics plotter devices used (§ 2.3), it has been kept separate to avoid the main simulation programs from getting too device dependent. By making a simple library of these routines they are now accessible to other programs as a plotting library. The plot post-processor programs being just one such application program based on that library to serve the SAMPLE simulation program's plotting needs. However, the power of this easily usable library is apparent when they are combined with the SAMPLE computation routines to display the (intermediate and all) profiles as soon as they are computed by the program. This *run-time graphics* capability was used on a couple of occasions to debug some problems in the profile advance routines, and the geometrical profile adjusting routines. By being able to quickly link them with the computation routines the problems were located and corrected with ease. And § 5.2 shows how such run-time graphics capability could increase the user-interaction power of the program.

Collecting the general routines into their own modules and then using the modules as higher-level abstractions for performing their tasks helps in the evolution and maintenance of the software, creates independently usable software products, and can have a synergistic effect when used with other software.

§ 5.9 Versions of the Software

For any piece of software undergoing continuous development, being worked on by different people, and probably being modified at different installations, numerous changes, big and small, occur even when starting from the same initial version. To well utilize the efforts put in making all these modifications and improvements in the software (documentation being

a part of it) it is usually desired to consolidate all of them together in the form of a new version. Before that can be achieved, merely keeping track of all the changes and finally extracting them becomes a major organizational and logistic problem for the software project. *

The term "version" connotes different things in different contexts. A new version implies a piece of software with some changes from the version it started from. To get some insight for controlling it, first it is necessary to keep track of the genealogy of any given version upto some commonly agreed upon point**. Assuming a general purpose programming language unhampered by any unduly constraining limitations of real computer systems (i.e. assuming fairly generous computing resources) a *reference version* of the software can be written. To install this on a real computer system some changes may be necessary. These are the inherent portability considerations like memory size, precision of numerical data storage and computations, access to facilities not defined in the programming language (e.g. date, time routines, or special file system access routines). With these changes to the reference version we get the actual installed site-specific or system specific version for the real programming environment of a system and peripherals ***.

It is important to have at least one up-to-date installed and executable version as close to the reference version as possible. This can serve as the *release version* when the program is to be ported to another machine (installation). No unnecessary system dependent code optimizations or tunings should be performed on this version to avoid having to undo them for a different computer system. The installed versions for different computers can be tuned to their computing environments for performance or local usage style reasons.

At UC Berkeley the local installed version of SAMPLE for general usage by the SAMPLE group is kept as free of system dependencies as possible*. Once every few months or a year, when it is relatively unchanging, the source code at that point is made the next release version of the program. By having the local version as close to the reference version (which exists only in concept and not in actual source form on the machine) and by making it the release version at specific times they all remain consistent with each other. Since the program identifies its version and the date and time of the run at the beginning of the output any bugs or undesirable behaviour can be traced to the version concerned.

---

* The discussion here is geared to the SAMPLE software project.

** In theory there is always the null point to start from.

*** So in Figure 4.1 the second level from bottom may be separated into two sublevels as (a) an *installation specific version*, below (b) the *reference version* of the code.

* This is the version on the "ucbesvax" machine, a VAX 11/780 running a 4.2BSD Unix system (and the f77 compiler). The source code is kept in the directory ucbesvax:~samsoft/release/ucb/ . The documentation and input/output examples are kept in different directories on the same machine.

It is intended that the release versions and the local general usage version(s) would form a chain i.e. a sequence in which every version except the current one has exactly one child in the sequence. But when any copies are made and modified (usually by different persons) the resulting tree of versions soon starts to grow in different directions. These diverging versions present a difficult organizational problem if any changes in two different versions from the same parent are mutually inconsistent or incompatible. The problem is not of merely keeping track of them (where tools like *sccs*, Source Code Control System [Roch75], on Unix can be very helpful) but that of merging them together to produce a single general version. Just adding user specifiable flags in the code to choose between different modifications, if they pertain to the same portions, produces a convoluted code which is difficult to debug, maintain or document for the user. The only way to avoid unnecessarily diverging versions, and to control and merge those which are necessary due to differing directions of the work (as opposed to different directions of coding) is to have a common vision and understanding for the development efforts and control the coding either voluntarily or administratively.

§ 5.10 Splitting the program into Stand-alone Simulated Machines

§ 5.10.1 Motivation

The SAMPLE program initially had only optical lithography simulation in it (Spring 1978). Yet when porting it to some minicomputers (PDP11/40 with RSX-11M, HP1000 with RTE-4B) substantial effort had to be put in to make it fit in the available memory of the minis using code overlays. With the addition of etching and deposition simulation to the program the memory size requirements grew even further. After the addition of e-beam lithography to all that it has become utterly impractical to even try to port it as a whole to small memory minicomputers. While super-minicomputers with virtual memory have been increasingly replacing less powerful minis, the rapid proliferation of powerful (16 bit) microcomputer systems at ever-decreasing prices makes it very attractive to be able to utilize them for running the program.

That alone is sufficient motivation to try to divide the program so that by running the simulation in parts advantage can be taken of these smaller but easily accessible computing resources. But an even more persuasive factor has been instrumental in prompting us to split the program: A natural subdivision exists in the program's simulation functionality as different physical processes, and the growth and development of the software in so many different directions has become increasingly more difficult to manage with all of it lumped together in one program.

Figure 3.2 (and its generalizations in Figures 3.3 and 3.4) shows the structure of the simulation task as the simulation of separate processing steps in a user specified sequence. If the user is concerned with only one of the processing steps, a quite common situation, all the other process simulation parts are not useful for these simulation runs. For the programmer(s), initially it is convenient to have everything together when a new simulated process is added to the program — that way all the required global variables and routines are easily accessible to the new code without worrying much about the details of communicating the variables from the old simulation steps to the new, and without the work of isolating the required common routines from the old code for linking with the new one. Slowly, even if the first program is designed with a good top-down and modular structure, the addition of the new code may perturb and erode that structure and eventually make it inconvenient to add any new code without violating the assumptions of the old one. This may occur because the function, structure, assumptions, requirements or details of these new tasks to be performed may be significantly different from those of the old tasks. For example, the assumption of an isotropic development mechanism with a development rate independent of the orientation of the surface normal in optical resists may be entirely unsuitable for many of the anisotropic etching rate mechanisms which affect the surface differently depending upon its orientation. So even though the overall control flow of the program is still very similar, and many of the same lower level support routines may be used for profile adjustment (checker, deloop of [OToo79], [Jewe79] and [Jewe77]) and the same output and plot routines used for displaying the profiles, the different rate functions and associated data requirements make the profile advancement code quite different. Trying to make all such differences special cases of a very generalized theoretical model, even if possible, has its own problems. It requires modifying the previously written and debugged code, and it may adversely affect its understandability, and the execution time and size requirements. (If it doesn't have any of these problems then the changes once made will quickly be "forgotten". But if it has any of these problems they will have a cumulative deteriorating effect on the software that could not be ignored for long.) Iteratively refining ("cleaning up") the code and with the perspective gained from the efforts, building better software modules for general usage (§ 5.8) seems to be the only practical approach in the absence of an *a priori* insight into the simulation and programming tasks.

At the current stage of the program * splitting it up into stand-alone simulated machines that use some of the common general purpose software modules of § 5.8 and share a common front end (user- or input-interface) module seems to be the proper way to handle the above problems and many others that arise when there are different programmers trying to extend it

---

* Version 1.5c which is essentially ver1.5b of May 1983 with some extensions.

to different simulated processes.

§ 5.10.2 The Clone-and-Trim Strategy for Splitting

When splitting the program into one simulated process per new program the conceptual integrity of the simulation or software philosophy and design should not be made to suffer. Otherwise, the changes will only produce a diverging version of the software that may not displace the original one, and hence lead to more problems than were there before. Maintaining continuity of goals and approach will ensure that no other useful changes will be hindered by the organizational changes made to the program for splitting. Therefore, after the splitting the users should still be able to simulate sequences of processing steps as in a real laboratory — the way it was possible with the all-together form of the software.

The adherence to the simple design of chapters 3 and 4, which is independent of the simulated machines being together in one program or each being a separate program by itself, and an implementation that preserves this simplicity makes it very straightforward to achieve the separation by a *clone-and-trim* strategy as follows.

Since the basic entity to be communicated between the machines is the wafer being processed, the problem is simply that of communicating the information representing the wafer (i.e. the state of the wafer) from one program to another. This can be achieved by writing that information to a file and then reading that file from the other program. So by providing the user with a pair of input statements to "put the wafer in a file named ..." and to "get the wafer from a file named ..." the problem of communicating the wafer with another program (even with itself) can be solved. After adding this put/get capability to the program that has all the simulated machines in it, it is a simple matter of making a copy of this program (i.e. *cloning* it) and then pruning out from this clone all the capabilities not essential to the simulation of process X and for wafer communication (i.e. *trimming* the clone) to get a stand-alone simulator for process X that can communicate with the original program. Repeating this procedure for each of the processes being simulated gives the desired separation of the original program into separate simulation-machines that can communicate the simulated wafer with each other.

It is not necessary to trim all the clones to have only one process per program. If desired, some clones may be trimmed only partially to retain the ability to simulate more than one process. The original program is simply a clone that was not trimmed at all (null trimming). And, in concept, the process of trimming can be reversed to join the different simulators together if name-conflicts and data structure incompatibilities are resolved in the code, and the control flow merged to satisfy the programming language requirements and the simulation design.

Making some adjustments in the software design would greatly facilitate the application of the above scheme to get the separate simulators. First, any unnecessary dependency in the input language for the program on any particular process to be simulated should be eliminated. Indeed this was one reason behind the cleanup of the input language syntax and processing to get the austere form described in § 5.5. (The enhancements suggested in § 5.7 do not affect this.) The software modules of § 5.8 are a good idea with or without this separation, and, of course, they help in the separation process because of the cleanliness and conciseness they induce in the rest of the code that uses them.

The put/get capabilities are easy to implement in a minimal form by not allowing the user any choice for the file name. That way the viability of the idea can be tested without struggling to change the current syntax of the input language which does not allow character strings as parameters following the initial keyword in an input statement. Later the syntax can be extended to allow user-specified file names in the input statements. From a conceptual view point, the files are but a representation of the (simulated) wafers and their names are essentially the names of the wafers allowing symbolic access to them from the input language — thus making the job of keeping track of different wafers undergoing different processes easier for the user.

With these simulated wafers existing outside the programs in the form of files used for communication from one program to another, the programs can be considered to have a four channel input structure as shown in Figure 5.3. The two channels (one for input and one for output) for the human user could be expanded further to the possibilities illustrated in Figure 5.2. And the two "wafer-channels" correspond to the communication link with the wafer collection in Figure 3.4 which could be a sophisticated data base when viewed from the simulated machines.*

That brings us back to the contents of the "wafer-files" communicated between programs. For simplicity, the first implementation is planned to have only the top profile of the wafer in the form of the coordinates of the (string-) points at some small intervals along the profile. This is adequate at present because even in the all-together form of the program only this string of points representing the profile is the main part of the data communicated between the simulated machines. (The general problem of representing the full state of the wafer does not seem to have a simple solution because of the different types and the total amount of information

---

* Details like the Monte-Carlo data file of § 4.4 and § 5.3 for e-beam lithography simulation are details of a different level and so of a different nature. Remember that the Monte-Carlo data file is associated with a subdivision of a computational step (as in going from Figure 4.3b to Figure 4.3c). Such files do fit in the general picture of Figure 5.2 but it is not necessary to twist the concept being presented in Figure 5.3 merely to accommodate them in that figure. It is not the intention to address such subdivisions in Figure 5.3.

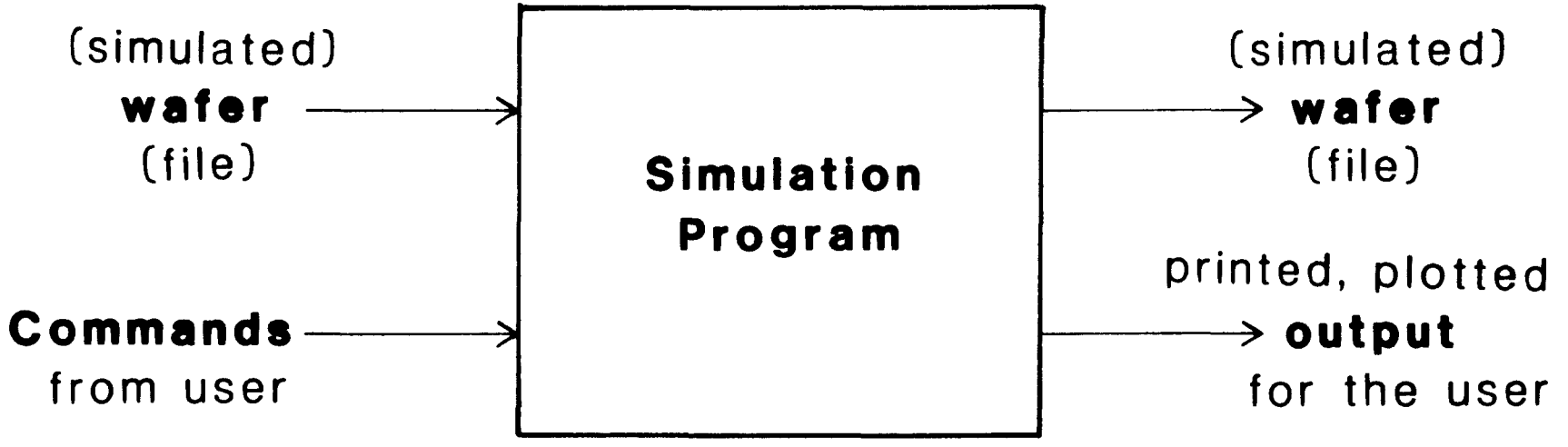


Figure 5.3 The uniform I/O structure of the programs for communication with the human user and with other programs

needed to be able to specify that.)

§ 5.10.3 Further Discussion of the Splitting

The software organization changes brought about by the splitting make explicit some aspects of the design which were not noticed when all processes were in one program. The plot postprocessor program which was separate for implementation reasons (§ 2.3) can be seen to be a simulated process — simulating the microscope (§ 3.2). It should read the wafer profiles from the same files that other programs use for communicating with each other. If profiles can be generated mathematically, or digitized from experiments, or constructed interactively at a graphics terminal that can be considered to be another "source of wafers" for the simulation.

Another example is the choice of default values, including the default geometry and profile used for the initialization of the wafer for each process. In the all-together version the optical lithography dominated the initialization because of the course of evolution of the program. This caused problems when the user wanted to simulate e-beam lithography, because it assumes a different wafer layer structure. So some patches were put in the program to override the optical layer initialization with the layer structure appropriate for the e-beam lithography simulation when the user invoked that process. For etching and deposition there was a related problem. They were to take their starting profile from one of the lithographies or the user could specify certain (piecewise linear) mathematical shapes by an input statement. They could also continue from the final profiles from each other. But to be consistent with the idea that there is always a default wafer (profile) present in the system, they needed a default profile in case no other simulation was run before them in the same invocation of the program. To complicate the matter further, the string of points that represented the (top) profile was not actually initialized in the lithography routines till the development process was started. So if the user wanted to start with the etching or deposition process (why? *) there was no starting profile present and caused (uninitialized variables and related) problems at the code level. Patches to set a flag to detect the presence of a lithography developed profile were put in the program to cope with this situation. But the real problem viz. the independence of the starting wafer profile on the process (or none) to be simulated, was not properly addressed by them.

This same point was missed when early in the project an input statement was added to "continue development" for optical lithography (say, for using a different developer, or plasma

---

* Whatever reason the user may have in mind, the program should have an overall consistent response. Having a default profile present is part of this consistency. Also, the etching and deposition machines point out the fact that the starting profile may come from somewhere else, some other process. This was the important point missed in the earlier design and control flow of the program because of the strong assumption of a flat starting profile to simplify the calculations required in the optical and e-beam lithography routines. The flat profile got "hardwired" into the program because of that.

ashing on the first profile). The profile initialization code (to a horizontal string of points) was skipped under the control of a "do not reinitialize the profile" flag rather than moving the profile initialization out of the development machine. Only when the idea of separating the program forced the profile to be "recognized" as an integral part of the wafer that is communicated from process to process were all these profile initialization or reinitialization problems clearly seen to be higher level problems and not just a situation to be patched at code level and documented with the corresponding awkwardness.

Among other advantages of splitting the program are that the smaller component programs can be understood, managed, and experimented on with greater ease by a person (or a small group). Because of their smaller size, hence smaller memory requirements, they will be portable to a larger range of machines encouraging more usage and development. By having a commonly agreed upon wafer-file structure, its contents and format, the programs need not all be written in the same programming language or even be present on the same computer system as long as the wafer files can be communicated with ease between the programs. And of course, being able to store and retrieve the state of the wafer (a partial state of the program) in a disk file itself makes splits in a processing run (i.e. applying a certain sequence of processes upto one point and then proceeding with slightly differing process steps thereon) easier to simulate without duplicating the common part of the simulation.

The decision to have only the top-wafer profile, not even the layer-geometry, in the files to be communicated between the simulation machines was made for the simplicity of implementing it. The profile is the main part communicated between the machines even in the together-version of the program. Even though that decision would imply that the user will have to duplicate some information (e.g. layer thicknesses, the physical and chemical parameters for the layers) separately in the (textual) input to the program this seems to be an acceptable price for the ease of managing the software. Further, the wafer-file could have a self-identifying header of one or a few lines that would identify its "version" (telling about its contents and format) so that if any more items of information are to be stored in it the programs would know about that additional information; thus allowing a systematic way for further software development.

Will the splitting cause more problems later on for controlling the versions of each of the separate simulated machines? It will certainly be easier to modify each individual program independently. But any such changes would not affect the other programs. And that is the main strength of this separated organization. Any changes that seem to get too difficult to manage for one program would be easier to discard, to fall back on a previous version of that program. So a "survival of the fittest" mode of evolution can operate more freely than for the together version. Any inherent design or technical problems that would come up or be noticed

would have been present in the together version too; because, in principle, the separate programs can all be joined together in one (§ 5.10.2). □

Chapter 6

Some Notes on the Methodology

§ 6.1 General

The preceding chapters have illustrated four major aspects in this software project: the computational resources (Chap. 2), the system configurations and major component processes to be simulated (Chap. 3), the path along which they are to be mapped on each other (Chap. 4), and the software itself — the considerations, facets, issues and solutions that make this a software engineering project (Chap. 5). This software project is part of an overall engineering research activity that also involves theoretical analyses of the physics and chemistry of the phenomena, and experimental investigations of the processes. The discussion of these latter two is outside the scope of this dissertation *.

The following sections bring together some points that were not mentioned in the earlier chapters in the interest of brevity or for maintaining the continuity of discussion. They are loosely arranged from general principles to specific points within each section.

§ 6.2 General Programming Principles and Practical Aspects

Correctness is the prime overriding property for a program. No compromises should be made regarding that. Correctness by design and construction should be one goal for the project. No bugs that are detected should be allowed to remain. Note that correctness of the code is distinct from, though it may be closely related to, the numerical accuracy limitations arising from limitations of modelling or of numerical methods (Figure 4.1).

Simplicity is a highly desirable quality. It helps the understandability of the software and hence helps achieve its correctness. The maintenance and growth are easier to manage when starting from a simple, minimal design.

---

* See the references at the end of part 1 of [SUG83].

§ 6.2.1 Some Misleading Sources of Numerical Errors Encountered in the Past

When constructing the software, its completeness at all levels (Figure 4.1) is easier to achieve if the programmer and the end-user use the same version of the program. The programmer should not have any extra tools or "hook-ups" in the code that are removed in the version available to the user. Mostly this is useful for tracking down any errors, or verifying their absence, for some specified input data. Many times an error at one level is wrongly attributed to another level (in Figure 4.1) by the end-user. Only by having diagnostic output capability (hook-ups) at all levels under scrutiny can the real problem be identified and the issue settled. (See also the example of Figure 4.4 in § 4.4.) In this regard it is interesting to note that every once in a while a user or user-programmer claims a need for "DOUBLE PRECISION" when some answers seem to be wrong. Every time this issue was raised regarding the SAMPLE simulations, careful investigation showed the problem to be somewhere else.

The first time this issue caused a serious concern was when the program was ported from CDC6400 system to a DEC-10 system (in Summer 1978). The CDC with 60 bits per word had approximately 14 decimal digits precision for the Fortran REAL variables. The DEC10 system with 36 bits per word had "only" about 7 significant digits for them. So when the developed resist profiles showed a few strange distortions the smaller word size and precision was thought to be the culprit. However, a dogged investigation (by John G. Mouton) revealed the source of the problem to be the difference in the range definition of the imaginary part of the complex natural logarithm function, CLOG, as computed by the two systems. The CDC Fortran library had a CLOG with a range of $-\pi$ to π radians for the imaginary part, while the DEC-10 Fortran library returned values in the range 0 to 2π. The manuals for both systems claimed their CLOG functions to be ANSI standard [ANSI66]. But their conformance to the Standard did not ensure the same computations on the two systems because the definition in the standard itself is incomplete regarding the range of the CLOG imaginary component. Once the problem was identified it was easy to adjust the range on the other system (DEC-10) to be as expected by the code i.e. to be the same as on the CDC6400 where the code was developed. Incidentally, later a different approach at the geometrical level (Figure 4.1) obviated the need for the CLOG function in the computations. (See [Jewe79] as compared to Appendix E of [OToo79].)

Another time this issue came up was after the program was ported to the VAX 11/780 (after Spring 1979). The program produced slightly different profile outputs when compiled with a debugging option to the compiler as compared to the output without that debugging option (in February 1981). Since the VAX with 32 bits per word has only about 7 significant digits of precision for the REAL variables, once again the smaller precision was thought to be the problem. The fact that the compiler produced machine code to perform the computations in double precision with or without the debugging option, but caused more intermediate

conversions of double precision quantities to single precision with the option on, seemed to lend credence to the claimed "need for double precision". This time a copy of the program was modified (by Claudio A. Fasce) to produce a "double precision version" where all declarations of floating point variables, constants, and functions were converted to their double precision forms *. The profiles from the single precision version and the double precision version had no significant shape differences so once again the claim of the "need for double precision" was found to be unjustified **.

There were a couple of other cries of "insufficient machine precision" raised that turned out to be uninteresting false alarms. However, there were three interesting situations (one of which has not been well-resolved yet) that were instructive.

The first of them was when it was observed (by Pradeep K. Jain, in early 1981) that by increasing the number of vertically placed grid points in the resist during optical exposure computations for a wafer with Aluminum substrate, the simulated time for developer breakthrough in the resist was drastically increased. At first, the code changes were suspected to be incorrect because the changes made in one module (the optical exposure simulation machine) were affecting another module (the photoresist development machine). But careful checking and experimental evidence of very long breakthrough times for photo-resist layers on Aluminum substrates showed that the difference was caused by the *increased* accuracy resulting from the larger number of vertical grid points! The strong reflectivity of Aluminum substrate caused strong nodes of light intensity in the photoresist where the development rate was extremely small compared to the rate in the well-exposed anti-node region (remember Figure 4.4 in § 4.4?). With fewer grid points the small node regions were not well covered by the grid, so the faster rates in the broader anti-node regions masked the effect of nodes in the profile advance computations. But with more grid points the node regions were better covered and hence the extremely slow rate there dominated the profile advance time — just the way it happens in a real lab. This experience once again impressed upon us the need for better numerical techniques to deal with the peculiar physical problems we are dealing with in this project.

The second example showed that the arithmetic precision was (ironically) more than enough for the computation. For simulating metal deposition at high temperatures, the original code of the deposition simulation machine (by Chia-Kang Sung) used a piecewise linear approximation to the Gaussian error function using interpolation from values stored in a table.

---

* ANSI standard does not define DOUBLE PRECISION forms for COMPLEX variables (though the local *f77* compiler on the VAX11/780 with Unix did allow it).

** Compilers on both these systems (DEC-10, and VAX11/780 with Unix) did not provide the user with any option to automatically convert all single precision floating point quantities and declarations to double precision. So the changes needed to introduce double precision had to be made by a tedious manual process.

The table only accommodated values within a three standard deviation (3σ) range for the function's argument. Beyond that the code abruptly truncated the function values to zero for the tails of the Gaussian curve. The values were integrated numerically for use in the simulation computation. When all this was replaced by an accurate algebraic approximation (by Ginetto Addiego) the resulting profiles showed pronounced wiggles in regions expected to be flat. This was diagnosed to be due to the instabilities caused by the accurate approximation that did not enjoy the "damping" effect of the tail truncation in the simpler approximation, so even minor disturbances propagated from nearby curved portion of the profile were growing larger rather than dying down. Finally the original less accurate approximation with truncated tails was left in the code as it was because overall it seemed to give sufficiently accurate profile shapes.

The third case (encountered by Prof. A. R. Neureuther) seems to involve the cumulative effect of the round off error in numerical computations. In optical lithography simulation when the development rate has very small and very large values in different regions (nodes and antinodes) of the resist, the program tries to take a large number ($>> 400$) of profile advances with smaller simulated time steps (i.e. far more than 400 points in the grid along the time dimension) to compute the final profile shapes. However that seems to produce much smoother profiles than those found in a real laboratory experiment. Limiting the total number of profiles (to about 400), causing correspondingly larger time steps, seems to produce satisfactory agreement with experimentally observed profiles. Because the program's original estimate of the total number of profile advances is supposed to be reasonably minimal while still maintaining sufficient accuracy in the computations, it seems very strange that a far smaller number of steps produces a better agreement with experiments in these cases. However, for lack of time and personnel resources, and because it seems to work, this problem has not been carefully tracked down yet.

These examples show (numerical) problems for which it is not easy to get good (accurate and fast) solutions. The numerical methods themselves are outside the scope of this dissertation. The examples were given to show the importance of keeping the software verifiable and testable at all levels (of Figure 4.1) at all times to properly diagnose any problems or concerns that may arise regarding its behaviour and output results. Often the optional diagnostic output is the most useful and convenient tool for such verification (see also § 4.4).

§ 6.2.2 An Example of Version Divergence

The story of the example above, of the Aluminum substrate and the vertical grid points, continues further. Instead of the brute force method of increasing the number of vertical grid points (with the corresponding increase in the run time and the memory requirements) a

different approach was tried (by Fasce in Spring 1981, and once again by Addiego in Summer 1981). Because the problem arises due to the shift in the placement of the grid points with respect to the position of the light intensity nodes, it was decided to locate the grid such that every node will have a grid point on it. This caused the two end intervals of the grid to be, in general, different from the interval being used inbetween. This solution was implemented (in two versions) and worked as expected. However, there is another simulated machine (the "diffusion" machine) that simulates the effect of post-exposure baking on the resist. That effect is modelled by a diffusion of the M-parameter species, and computed by a two dimensional (2D) convolution of the M-parameter matrix, corresponding to the 2D placement of the grid points, with a 2D "impulse response" function. The non-uniform spacing of the vertical grid points would require a much longer computation to perform the convolution in this simulated machine *. For that reason, and because the "diffusion" simulation is not used often, that code was not modified to handle the different grid intervals. As a result the shifted-grid solution has not been integrated into the local standard version of the SAMPLE software. And as more and more small modifications accumulate in the local standard version it diverges further and further away from the code implementing the solution — a version divergence problem (§ 5.9) that does not seem to have a clean and easy solution.

§ 6.2.3 Portability and Related Considerations

Easier access to the program, by being able to use many different computer systems (computing resources) was one of the main goals of this software project *. This desire implied that the software be written in a widely available language, be independent of any peculiar hardware features or special software libraries, and be portable as a whole to another site having a reasonable amount of computing resources (§ 2.3, § 5.2 and § 5.3).

Accordingly, the software was written in standard Fortran IV [ANSI66] and is portable to any computer system with a standard Fortran compiler and sufficient memory. The few deviations from Standard Fortran IV, for file handling and character string manipulation, are coded in standard Fortran 77 [ANSI78]. The access to the system specific routines (e.g. date and time functions) is through clearly isolated routines that are easy to modify or turn off when porting to a different system. All this has resulted in a portable set of simulation tools that are being used at numerous installations. The feedback from all this usage has contributed many

---

* This 2D convolution can be performed as two 1D convolutions because the effect of the Gaussian function used as the "impulse response" is separable in the two orthogonal directions of the axes due to the linearity of the convolution operation [Nand80]. Even with the resultant large savings in the computations the effect of different grid interval sizes in the vertical direction would be a despairingly large computation time and memory requirements.

* In part this was due to the nature of support (funding) for this project in the early stages.

valuable ideas to the software and the project.

While the goal of portability has proved to be an important one, it is useful to be aware of the restrictions it has imposed on the programming effort. More modern programming languages like Pascal [Jens75], C [Kern78], or Fortran preprocessors like Ratfor [Kern76] have been widely available for many years now. The richer control structures and data types provided by them, along with the freedom from many lexical restrictions, increase programmer productivity and would be very helpful for programming the non-numeric parts of the software dealing with user-interaction and file-system interfacing. Even though Fortran can be used to get the task done, as all this software readily demonstrates, constraints imposed by Standard Fortran are often felt as a dragging force on the growth of the software.

A large size is an hindrance to portability (§ 5.10.1). It prevents the user from utilizing the plentiful resources of the small computer systems. But it would be a mistake to distort a clean organization of the software and code to simplify overlaying of the code on minicomputers. Such distortion may occur if "portability" to such computers is considered necessary enough for the software. For quite some time the program did not have any major modules other than the user interface and the individual simulated machines. Many common routines (e.g. geometrical routines for profile adjustment, routines for producing plots on the lineprinter, or some mathematical or numerical routines) were not grouped together to form general "library" modules — partly in order to avoid the trouble of linking such modules, called from various machines, into one-level overlay tree structure required for some minicomputers (e.g. the HP1000 with RTE operaing system, 1979) *. However, with the further familiarity and understanding gained over time such general purpose library modules are being formed to clean up the code and its organization.

Another aspect of portability is the method and medium used for transferring the software from one installation to another. On one occasion we used punched cards to transfer the program to a minicomputer. Recently, transferring it electronically over a computer network within campus and to some outside installations has become possible and convenient. But the most widely used medium is the magnetic tape **. The storage capacity of commonly available tapes (1200 feet or longer) is large enough to hold all the source code of this project along with

---

* Another reason for not forming such libraries earlier was that the small variations in the copies of such routines used by different programmers working on different machines were difficult to reconcile with each other because of a lack of good communication between them and due to the looseness in the group's organization. (There was a time when four persons working on the program were all using different systems on campus to edit and store the source code without a convenient means of transferring it from one system to another.) Because of the fluidity of the code being worked on by a person, and its rigidity when the same person was not available to work on it (for whatever reason), it was difficult to bring together some of the diverging versions of the sharable modules.

** The floppy disk may soon become another viable option on a smart terminal or personal computer.

the documentations, many input/output examples, and instructions for installation, all on the same tape. The format chosen is usually a tradeoff between ease of writing the tapes on the sender's (our) system and the ease of reading it on the recipient's system which is often of a different type. To help the recipient read the many files off the tape in a convenient and systematic manner, putting some *bootstrap* files on it is a good idea for a *release-tape* design. A bootstrap file may contain the computer commands (if the sender is familiar with the recipient's computer system) that can be read into a command script file to be executed to read (the rest of) the tape. Or it may simply contain a list of the files on the tape so that the recipient may edit it to generate a suitable command script file to extract the other files off the tape. Having such a command script file for the system on which the tape was originally written is helpful when reading it back in to verify its completeness or to diagnose any problems regarding it. A command file for compiling and linking the source code is another of such useful tools *. And a well-designed and complete "release tape" will be equally valuable as a backup or archival storage tape for the project.

§ 6.2.4 Design for Growth and the TRIAL Statement

The users of a program are a valuable source of ideas. Their collective insight can be tapped by making it easy for them to shape the program to their needs and desires. When the users can modify a program, experiment with it at all levels (of Figure 4.1) and enhance it without going through an external agent, the programmer, a triple advantage is accrued: the users get what they want, the "programmer" is not burdened with all the work, and the program acquires the enhancements that make it more useful for other users.

The TRIAL statement in the input language to the SAMPLE program was introduced with these very ideas in mind. It implements the essence of user interfacing by simplifying it to a minimal form. To use it, the user enters the keyword TRIAL followed by some numerical parameters. The program collects the parameter list in an array and calls a routine to execute (interpret) the TRIAL statement. By convention, the first number (which must be present in the statement) indicates the action to be performed with the rest of the parameters, if any (§ 5.5). The user can put in the code that will check the action number and perform the action corresponding to it. The state-variable model for the program allows all actions to be treated as parameter setting or activation of a state-transformation procedure ** (§ 4.3). Thus it is easy to integrate a new action into the program.

---

* See also [Gris82] for some good ideas on release tape design.

** Combinations of parameter setting and state-transformation activation are not needed and are not encouraged in order to maintain a simple and non-duplicated set of activation commands.

The simplicity of this arrangement combined with the easily comprehensible user-models of the simulated system (§ 3.2, § 4.2, and § 4.3) and the modular structure of the program (§ 5.8) have resulted in an overwhelming contribution to the functional capabilities of this simulation tool by various persons. As the word "TRIAL" indicates, this was originally meant to let the users try various things on a TRIAL-and-error basis — to be filtered out and incorporated in a "permanent" manner later by a "programmer". But soon after its introduction, the number of (actions performed by) TRIAL statements far exceeded the number of originally provided statements (actions). And by a straightforward macro-like mapping of other keywords to the keyword TRIAL and an action-index (§ 5.6 and § 5.5), now all input statements are treated in the same manner as the "TRIAL statement". This has been an instructive lesson in simplicity.

The introduction of character strings in the input (e.g. for filenames (§ 5.10.2)) or any of the input language enhancements considered in § 5.7 do not change the nature of this TRIAL-statement concept*. The input statements with their various parameters are only the means used by a user to interact with the simulated laboratory, just as the commands with all their command-line arguments are the means used by a computer user to interact with a computer or an operating system **. And the less artificial the distinctions are between the "system-provided" utilities and the users' contributions (as in Unix), the more responsive will the evolution of the system be to serve the users.

§ 6.2.5 Handling Multiple Wavelengths in Optical Lithography

The initial versions of the optical lithography were for single wavelength illumination. Then they were generalized to handle multiple wavelengths. Two points regarding this generalization are noted here.

First, the proportion of incident energy (dose) at each wavelength is, in general, different for every point in the aerial image of the mask (i.e. the image intensity curves for individual wavelengths are not related to each other by a simple proportionality factor). Because of this, a separation of the "bleaching" computation into standard exposure and actual exposure (see Figure 4.3) is not strictly correct for the multi-wavelength case. However, this separation, which allows an efficient computation of the exposure simulation process, was maintained by introducing a new approximation. The standard exposure computation is performed assuming the same proportion of energy (the weights for the wavelengths) as in the incident illumination

---

* In particular, any future extensions for providing control structures in the input language to allow compound statements or actions (e.g. looping commands) should be built upon the simple unit-actions of these (TRIAL-)statements as discussed in § 5.7.

** This analogy also holds for the newer styles of user interaction with menus, windows, or whatever (§ 5.4).

(as if there were no mask present). Then the actual exposure of a column of resist below a horizontal grid point (on the top surface, for which the aerial image is computed) is computed by interpolation on the standard exposure computation (the columns of the matrix) with respect to the total image energy incident at the top surface. This amounts to approximating the component curves of the image intensity, at the different wavelengths, by curves proportional to their sum, with the proportion being that of the no-mask (i.e. wide open space) case. This approximation is justified by the smallness of the horizontal shift of the steepest part of the image components for a mask edge, and the good agreement of the simulation results with actual laboratory experiments. But when simulating thin lines or spaces whose widths are close to the horizontal shifts between the steeper edge images this approximation may be in serious error. Let the user beware.

Second, the original input statements for entering the wavelength related information (the LAMBDA, RESMODEL, and LAYERS statements) cannot handle all the information needed for the multi-wavelength case. The two statements added for the multi-wavelength case (TRIAL 21, and TRIAL 31) supersede the original input statements for wavelength information but they do not address the layer thickness information. So the user is forced to use the old statements for specifying the thickness information for the wafer layers, and then supersede the wavelength related information by the latter statements. A cleanup of the input statements could be done by introducing a new statement for specifying the thicknesses (only) of the wafer layers and then eliminating the old statements (RESMODEL and LAYERS). This cleanup would prevent the confusion in the new user's mind about the proper sequence for entering the "old" and the "new" statements *.

These examples **, of how a generalization (from single wavelength to multi-wavelength illumination) at a top level (Figure 4.1) affects the previous work below it, may be applicable once again if the other lithography processes being simulated are generalized in a similar manner.

§ 6.2.6 Miscellaneous Notes

An interpretation aspect of Figure 4.1: The plasma ashing process which removes materials at uniform isotropic rates can be simulated by using the etching simulation machine with isotropic rates, or using the (optical) lithography simulation's development machine with its rate parameters adjusted to give a constant (i.e. uniform and isotropic) rate. Similarly the

---

* Because of inertia, this cleanup has not been done yet. Also, this example shows that an input statement should address only one type of information.

** See also the discussion of image and OTF plot-data outputs for the multi-wavelength case in § 5.3.

Chemical Vapour Deposition (CVD) process may be simulated using the etching simulation machine with a negative isotropic rate. Thus the same routines may be interpreted as simulating different types of physical machines. Incidentally, the deposition simulation machine is tailored to simulate various metal deposition machine configurations used in a real laboratory and cannot handle the isotropic rates needed for CVD.

Using floppy disks as wafers: Many workstations or personal computers with their own removable media storage are now being used to communicate with each other and with larger machines. This gives a new tangible feel for the wafer files of § 5.10.2 or the wafer collection of Figure 3.4. By storing these wafer files (simulated wafers) or wafer collections on floppy disks that can be conveniently carried around or sent to another place it will be practical to maintain a good communication between different simulated processes on different computers at different sites. This may have the same effect on the usage of such simulation programs that specialized processing at different laboratories can have on the processing of a batch of real wafers.

Human readability of all communication files: The use of a human readable form for the communication files is to be preferred over any machine dependent binary encoded formats. The latter may make efficient use of storage and processor time for reading and writing them, but the former are conveniently communicated over any ordinary communication links between different computers (e.g. when using ordinary electronic mail) and are easy to diagnose if any mistakes occur or are suspected.

Documentation structure for the "User Guide": There should be different types of documentation for the program corresponding to the various models given in Chapters 2 to 5. Figure 4.1 covers most of the anatomical structure of the software. The User Guide may be arranged such that the topics in each layer are covered in a sequence (one layer after another) starting at the top layer. Then the users can read it up to their depth of interest in the program. This was one of the main considerations in the organization of the SAMPLE User Guide [SUG83]. For a "programmer" who is going to install the program on another computer system the relevant documentation of the lower most level is the "installation guide" (including the description of the "tape" (§ 6.2.3)). Documentation and its structure is an important consideration because if some point is difficult to document or explain then that could be a strong indication of some awkwardness in the software or its design that should be tended to.

Which way is up? : The need for the two types of documentation — "User Guide" oriented towards users interested mainly in the physical laboratory systems and their simulation, and "installation guide" for the programmer concerned with utilizing the available computing resources, merely symbolizes the two basic extremities in the software design efforts. These are the two ends, the top and the bottom layers, of Figure 4.1. The software project aims to

bridge these two with the user as the implicit third reference point (for whose use the software and the documentation are written). While the design of the user interaction and the input language (§ 5.3, § 5.4) seems orthogonal to the vertical dimension of Figure 4.1, for the software project as a whole there is a constant movement along a three sided ladder of the functional capabilities of simulation of different physical processes, the utilization of the available computing resources, and the interaction of the user with these two (the user interface). The software project moves ahead by moving forward along any of these three sides. At every step there may be newer tradeoffs to be made between them. Depending upon the difficulty of the problems encountered when moving along these three sides, the focus will be shifted to make one of them the dominant or the "up" end. Any equilibrium position approached is soon disturbed by a desire for more or the movement of one of the three sides: newer physical processes to be simulated, newer or previously unused computing resources to be used, or trying to make it more convenient for the user to use. During the course of the project, if this constant shift and exchange of the "up" end is not kept in the proper perspective it may distort a "top-down" structure of the software. And the software may acquire many appendages that were very useful at the time they were added but are not that valuable any more.

Is there a "pure programmer" counterpart of the "pure user"? *: In a project of this size, this question arises in various forms and degrees, particularly if the tasks from the different layers of Figure 4.1 are to be divided between different persons. Though rough divisions can be made as user, analyst, programmer (going from top to bottom of Figure 4.1), the boundaries between them are quite mobile and often tend to get very fuzzy. Without good communication between them a good vision of the overall project may not be present in the group; which may result in a lot of mutually incompatible activities without a common fruit. There is no "pure programmer" for a project like this. But this is hardly surprising because there is no "pure user" either. The substitution view of § 2.2 is an ideal that can only be approached but may never be fully attained; thus leaving the users aware of the simulation nature of their activity.

§ 6.3 Frustrations in a Group Project

The SAMPLE software could not have progressed through all its useful states and acquired its current useful capabilities without the contributions of many different persons who worked on, used, evaluated, criticized and continually improved it as a collective effort. And the loose organization of the group has many times provided useful freedom from undue constraints and formalities. Yet there are frustrations that may be noted down.

---

* See the discussion of classification of users, in § 4.4.

First, in an University environment, working on a project one of whose main goals is to consolidate the physical insights into generally usable program leads to a constant tension between "research" work and "product" work. It is worsened when the sharing of these tasks between different persons is in very different proportions (for whatever reasons)*.

The second frustration, related to the first, is when decisions regarding the software and documentation management cannot be enforced (or are not enforced) for reasons other than the technical points involved. Again this is particularly damaging for the group when the wasteful consequences of not implementing the decisions affect (or get distributed among) the members in different proportions. Simple examples of this can be seen when obsolete or diverging versions of the code or documentation are not purged after the common standard version has been carefully updated and verified, or when the administration of the project fails to implement the necessary decisions because they were delegated without delegating the authority necessary to enforce them **. The obsolete versions pop up unexpectedly (or worse still when they pop up dreadfully expectedly) and invalidate a lot of the work put in the previous updating. When more than one person contributes to it (because of lack of communication or poor project administration) undoing the bad effects has a significantly deteriorating effect on the general morale and group unity. Most of such compounded effects could have been avoided at hardly any cost by not being complacent about letting obsolete versions remain accessible and by not being complacent about unjustified claims of ignorance or of lack of confidence in the updates and cleanups done by others.

Perhaps all this is a part of the social process within whose context a research project operates. From a detached (academic?) viewpoint, even the problems leading to these frustrations make an interesting study. □

---

* All the talk about the proof of the research pudding being in showing a piece of code that "works" is not much of a guideline when the tasks of devising the different recipes, actually cooking them, and then eating the working product are shared in different proportions by different persons.

** "Feedback" is not of much consequence if it is not used to correct the error.

Chapter 7

A Retrospective and Directions for Future Work

§ 7.1 General

The SAMPLE project is still growing and evolving in different directions and at different levels. The designs, plans, significant decisions and observations shaping this project have been presented in this dissertation to serve as an exposition and documentation of its structure. Because of the planned continuity of this work some of the decisions and actions in the past are geared towards things yet to be done in the future. In this final chapter we take a look back on this dissertation (§ 7.2), and on this project (§ 7.3), and then towards the future in an effort to outline the directions of future work (§ 7.4). Naturally, there is a considerable overlap between them.

§ 7.2 Summary of the Previous Chapters

Matching the computational resources to the simulation needs and desires has been the theme of this software project (§ 1.1, Chap. 4). Its goal was characterized as the substitution view in § 2.2. Its approach of first formulating a view of the processes to be modelled and then systematically mapping them through various levels to the computational resources was described in Chapters 3 and 4 respectively. Chapters 2 and 5 examined the computational resources, and the software approaches to get the most out of them. Chapter 5 also described the process of building the software in some general ways that are particularly relevant to this project. Chapter 6 continued in the same vein to bring together Chapters 3, 4, and 5 within the common context of this simulation project. Together they document the many aspects and details of this software — its structure, environment, context and its growth process.

§ 7.3 A Retrospective on this Software Project

In the evolution of this software it has been constantly observed that the solutions obtained are always with respect to the problems addressed, the resources, constraints and the stage in the evolution of the project. As some problems get solved the relevance or importance of others at the same or different levels (Fig. 4.1) is often significantly changed. Only a good

overall understanding, e.g. in the form of models as shown in Figures 3.4, 4.1, and 4.2 can help maintain a proper perspective over this constantly changing landscape.

Neither the software nor this dissertation or the concepts and models presented therein emerged top-down, or by any *a priori* process, in their current form. A large number of iterations of "design - implementation - use - evaluation - redesign" shaped the software, the documentation, and their overall implementation in their current state. The models and approaches learned in the process have been presented in this dissertation. They should be useful for understanding the reasoning behind what has been done and for continuing it. *Even if* a totally different approach is adopted later, this work would serve well as a concrete example for comparison with the future work and ideas coming up. Also, by putting the ideas in a written form, this dissertation should be useful for evaluating the current design by itself.

One simple attempt for judging the significance of this work is to answer questions like "What has been learned from all this?", "What are the results?", "What is the product of all this effort?" The answer to the first question is given in the paragraphs above and detailed throughout the previous chapters. The results and the product have been the understanding of the problems, the understanding of the ways of solving them and the solutions obtained. On a more pragmatic level the software itself is a visible product along with its usage as a learning and investigation tool by students in IC courses and research workers in the field *.

Two other questions, sometimes encountered in slightly modified forms, may be addressed here. Question 1: "How would you do it all over again with what has been learned in doing it by now?" The answer would be simply: By applying the models and principles outlined so carefully in the previous chapters. If that seems to lead to a product quite similar to what has already been done it is only because the ever-present iterations of "design - implement - learn - redesign" have brought the current work as close to the understanding gained as is possible within the current working environment. And question 2: "If you had your choice what programming language facilities and computing resources would be useful for this work? If you could design them yourself how would you do it?" This question partially inverts the constraints of the form "available resources" by the "freedom" (problem?) to design them. The obvious answers are faster CPUs, larger memories, and the wish for better graphics interaction capabilities, and somehow (how?) more convenient methods of interconnecting software modules. But the detailed answers would emerge only by studying how, in Figure 4.1, the original simulation problems are getting mapped onto the computing resources. Essentially this question only pushes a "fixed boundary" of the original problem from "computational resources" to their design and the technology of building them as mastered by the problem

---

* See the list of publications at the end of part 1 of [SUG83].

solvers **.

§ 7.4 Directions for Future Work

Many areas for further work are noticeable in the description and discussion in the previous chapters. However, some directions that are not addressed by the software at present are not mentioned there. This section presents many directions of future work at various levels in this project.

For the immediate future, the program should be split up and a format should be defined for the communication files (§ 5.10). Further, all the simulation processes (simulated machines) need a better documentation at various levels (of Fig. 4.1). The User Guide [SUG83] gives many details for using the program but there is no good documentation of some of the crucial details like the discretizations used by the program. A systematic top-down (in Fig. 4.1) documentation of the simulation machines and the general code modules would be helpful for the persons working on the code, and it may also reveal some ideas for cleaning up or refining the software. The need for such documentation is apparent every time some such detail or the intention of a previous programmer who had worked on that part of the software in the past has to be deciphered by reading the code.

Another item that should be addressed soon is the refinement of the current "release tape" structure to include the "bootstrap" files and other tools to make it a good "archival tape" for this software (§ 6.2.3).

For the longer range planning of the project many extensions of this project may be undertaken. The simulation functionality could be extended to multilayer resist techniques, investigation of proximity phenomena, two dimensional mask patterns and many other problems that other members of the SAMPLE group are more conversant with.

From the programming side, the details of the wafer-files (§ 5.10.3) could be refined and extended to handle more information about the wafers without losing the simplicity of a minimal format. A code module of routines for easy access to these files should be written. Input language extensions should be designed to make the data abstractions more transparent to the user. The user should be able to say "for wafer *xyz* perform the processing steps of ... ". Many parts of the code should be converted to be "data-driven" so that the code can be generalized and managed more easily by the programmer(s). This would be particularly helpful for the input interface module (§ 5.6). Run-time graphics capabilities should be introduced to make the interaction with the program more direct for the user (§ 5.2, § 5.8).

** See also the ideas of Prof. Kenneth G. Wilson of Cornell University, Ithaca, New York, U.S.A. [Wils83].

The database for the wafers (the wafer collection of Figure 3.4) could be extended to include the properties of materials, and even the default settings or some other preferred settings of the processes. That would make it easier to adjust them, and to specify the wafer components directly by specifying the materials rather than their individual physical properties and parameters like refractive indices, development or etching rates etc. It would also make their specification more self-consistent. Probably this would involve some kind of a relational database structure because the wafer materials as well as the process parameters, e.g. wavelength used for optical lithography, and components, e.g. the developer being used, together determine the values of parameters like refractive indices and development rates; and it would allow the user to change the wafer materials or the process components with an input specification at least as simple as one in a laboratory process specification sheet. *

More ambitious plans regarding simulation functionality and its programming may be undertaken in attempts to join the current software with other process and device simulator programs. Similarly, the simulation software may be joined with other experimental data collection software. In this regard Figure 7.1 and its generalization in Figure 7.2 show a grand map for the construction of a software network from different programs that approach the various IC engineering problems from different sides (experimental, computational, or theoretical). Though this would be a gigantic undertaking, sketching it out would give many ideas for attacking parts of it within an overall coherent perspective. A start can be made by finding meaningful abstractions and their implementation designs for interfacing the different programs within the fields shown in those figures. Even with a minimal interfacing such combinations of programs would be powerful application tools within their domains of applicability.

Another useful type of software linkage would be to wrap this simulation software with an optimization oriented software package like the DELIGHT system developed at UC Berkeley [Nye83]. After introducing the proper communication links to such software, the optimization package could be used to systematically search for a combination of parameters to achieve some specified objective that has been expressed using a variety of conditions on the results of the simulated processes.

How many of the above directions will actually be pursued will probably depend on the many factors faced by the people involved. Other directions may open up in the course of following any given direction, and serendipitous results may also reward the efforts. As a planned engineering project with a continual feedback and the freedom to change a few goals along the way many functionality aims would undoubtedly get set to solve problems as they are encountered or noticed. Perhaps the only constant part of the structure may be the existence of a

---

* See also [Reid84].

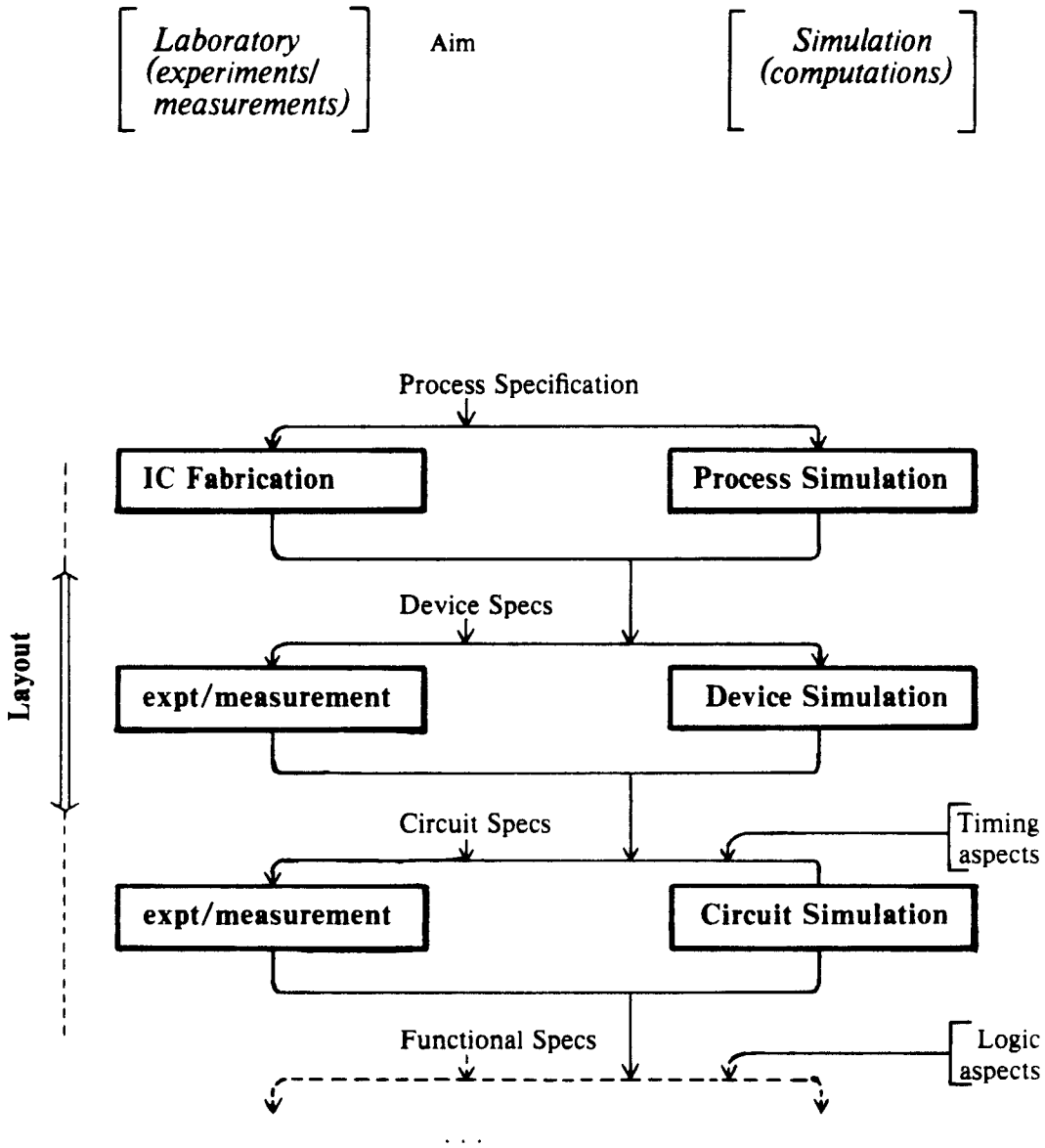


Figure 7.1 Combining simulation and experiments for a range of IC electronics fields

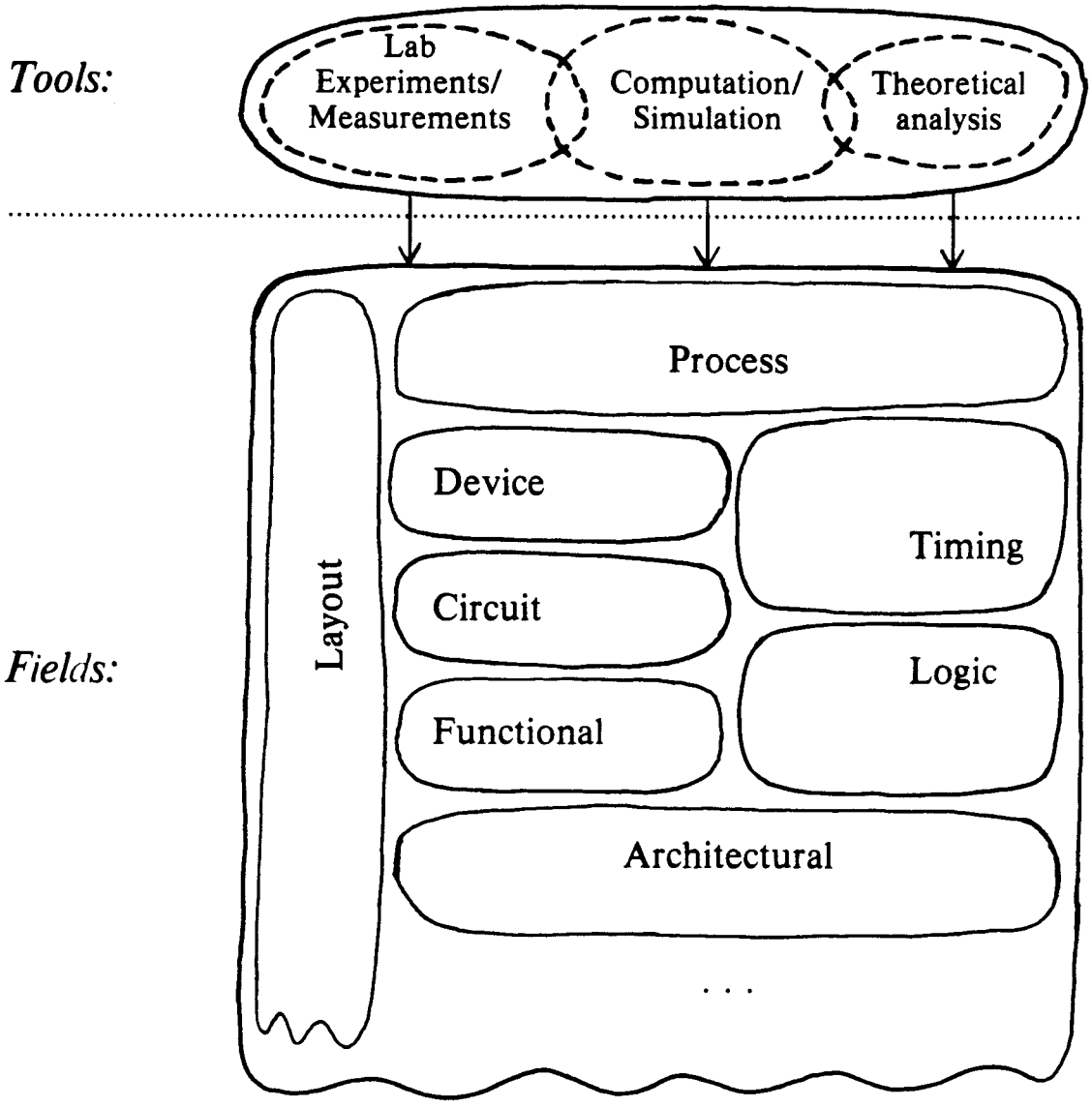
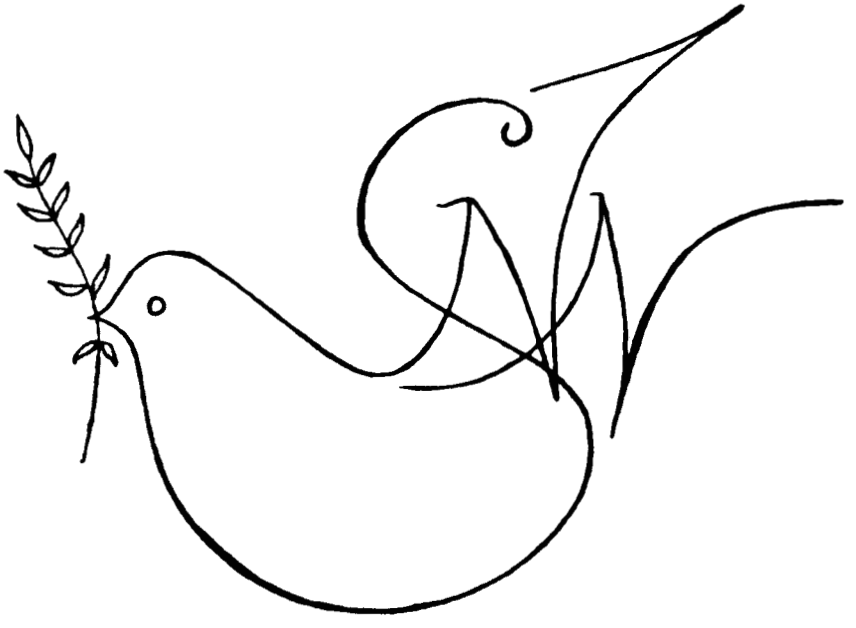


Figure 7.2 A diagram showing the tools and fields in IC electronics engineering

mapping from real phenomena to computations. That is necessary if the project is to stay relevant in the engineering sense. Yet there would always be many improvisations within the mapping depending on the people performing it. Similarly, the design of the user-interface would always be dictated in its function by the motif of imitation (§ 2.4) and in its capabilities by the available skills and resources. But its style would be determined by the people — the users (§ 4.4), who are an integral part of the whole process. For it is the usage of the software product that will determine its course of evolution, shape its being, and prove its worth.

Dream of plans and force them into implementations. Whatever the results, analyze them and continue on to the next dream.

□



References

Aho77. A. V. Aho and J. D. Ullman, *Principles of Compiler Design,* Addison-Wesley, 1977.

ANSI66. American National Standards Institute, Inc., "American National Standard Fortran," ANS X3.9-1966, New York, 1966. The ANSI standard Fortran IV. See also the clarifications in *Communications of the ACM*, vol. 12, no. 5, pp. 289-294, May 1969; and vol. 14, no. 10, pp. 628-642, October 1971.

ANSI78. American National Standards Institute, Inc., "American National Standard Language FORTRAN," ANSI X3.9-1978 (Revision of ANSI X3.9-1966), New York, 3 April 1978. (The ANSI Standard Fortran 77.)

Anto79. D. A. Antoniadis and R. W. Dutton, "Models for Computer Simulation of Complete IC Fabrication Process," *IEEE Transactions on Electron Devices*, vol. ED-26, no. 4, pp. 490-500, April 1979. (A thorough discussion of the SUPREM Program.)

Arno81. K. C. R. C. Arnold, "Screen Updating and Cursor Movement Optimization: A Library Package," in *Unix Programmer's Manual (for the 4th Berkeley Distribution)*, June 1981. Documentation for the *curses(3X)* library on Berkeley Unix. This package was written around early 1980 or before.

Dahl72. O.-J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, *Structured Programming,* Academic Press, 1972.

Gris82. R. E. Griswold, "A Tool to Aid in the Installation of Complex Software Systems," *Software — Practice and Experience*, vol. 12, no. 3, pp. 251-267, March 1982.

Jens75. K. Jensen and N. Wirth, *PASCAL User Manual and Report,* Springer-Verlag, 1975.

Jewe77. R. E. Jewett, P. I. Hagouel, A. R. Neureuther, and T. Van Duzer, "Line-Profile Resist Development Simulation Techniques," *Polymer Engineering and Science*, vol. 17, no. 6, pp. 381-384, June 1977.

Jewe79. R. E. Jewett, *A String Model Etching Algorithm,* M.S. Project Report, University of California, Berkeley, California, 18 October 1979. Also available as ERL Memo No. UCB/ERL M79/68.

Kern76. B. W. Kernighan and P. J. Plauger, *Software Tools,* Addison-Wesley, 1976.

Kern78. B. W. Kernighan and D. M. Ritchie, *The C Programming Language,* Prentice-Hall, 1978.

Lee83. K. Lee, Y. Sakai, and A. R. Neureuther, "Topography-Dependent Electrical Parameter Simulation for VLSI Design," *IEEE Transactions on Electron Devices*, vol. ED-30, no. 11, pp. 1469-1474, November 1983.

Nage75. L. W. Nagel, *SPICE2: A Computer Program to Simulate Semiconductor Circuits,* Ph.D. Dissertation, University of California, Berkeley, California, 9 May 1975. Also available from UC Berkeley, Electronics Research Laboratory, as Memo No. ERL-M520.

Nand78. S. N. Nandgaonkar, *Report on the Design of a Simulator Program (SAMPLE) for IC Fabrication,* M.S. Project Report, Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, California, 22 June 1978. Also available as ERL (Electronics Research Laboratory) Memorandum No. UCB/ERL M79/16.

Nand80. S. N. Nandgaonkar, *(A private note on the diffusion machine to the SAMPLE group)*, Originally probably before July 1980.

Nye83. W. T. Nye, *DELIGHT: An Interactive System for Optimization-Based Engineering Design,* Ph.D. Dissertation, University of California, Berkeley, California, June 1983. Also available as ERL Memo No. UCB/ERL M83/33.

Oldh79. W. G. Oldham, S. N. Nandgaonkar, A. R. Neureuther, and M. M. O'Toole, "A General Simulator for VLSI Lithography and Etching Processes: Part I — Application to Projection Lithography," *IEEE Transactions on Electron Devices*, vol. ED-26, no. 4, pp. 717-722, April 1979. This was a joint special issue on VLSI, also published in *IEEE Journal of Solid-State Circuits*, vol. SC-14, no. 2, April 1979.

Oldh80. W. G. Oldham, A. R. Neureuther, C. Sung, J. L. Reynolds, and S. N. Nandgaonkar, "A General Simulator for VLSI Lithography and Etching Processes: Part II — Application to Deposition and Etching," *IEEE Transactions on Electron Devices*, vol. ED-27, no. 8, pp. 1455-1459, August 1980. This was a joint special issue on VLSI, also published in *IEEE Journal of Solid-State Circuits*, vol. SC-15, no. 4, August 1980.

OToo79. M. M. O'Toole, *Simulation of Optically Formed Image Profiles in Positive Photoresist,* Ph.D. Dissertation, University of California, Berkeley, California, June 1979. Also available as ERL Memo No. UCB/ERL M79/42.

Reid84. B. K. Reid, J. D. Shott, and J. D. Meindl, "Wafer Fabrication and Process Automation Research at Stanford University," *Solid State Technology*, vol. 27, no. 7, pp. 126-133, July 1984. Integrating Lab and Computers for the specification of fabrication processes (for process control in the lab and for simulation). Applying programming language concepts.

Roch75. M. J. Rochkind, "The Source Code Control System," *IEEE Transactions on Software Engineering*, vol. SE-1, no. 4, pp. 364-370, December 1975.

Rose81. M. G. Rosenfield, *Simulation of Developed Resist Profiles for Electron-Beam Lithography,* M.S. Project Report, University of California, Berkeley, California, 12 June 1981. Also available as ERL Memo No. UCB/ERL M81/40.

Ryss80. H. Ryssel, K. Haberger, K. Hoffmann, G. Prinke, R. Dümcke, and A. Sachs, "Simulation of Doping Processes," *IEEE Transactions on Electron Devices*, vol. ED-27, no. 8, pp. 1484-1492, August 1980. (Contains the theory behind, and usage examples from, the simulation program ICECREM.)

Selb80. S. Selberherr, A. Schütz, and H. W. Pötzl, "MINIMOS — A Two-Dimensional MOS Transistor Analyzer," *IEEE Transactions on Electron Devices*, vol. ED-27, no. 8, pp. 1540-1550, August 1980.

SUG83. *SAMPLE User Guide (Version 1.5b, May 9, 1983) Parts 1-4,* Electronics Research Laboratory, Department of Electrical Engineering and computer Sciences, University of California, Berkeley, California, May 1983. Available from ERL, UC Berkeley, as Reports SAMD8 (for Parts 1-3) and SAMD9 (for Part 4). Updated User Guide for the forthcoming version 1.6 is in preparation (Summer 1984).

Wils83. K. Wilson, Probably early 1983. A public talk at UC Berkeley, in the Department of Electrical Engineering and Computer Sciences. He talked about his ideas of building a workstation for scientific and engineering analysis with emphasis on numerical computations. One of his points was that the scientist should be able to specify the equations, the numerical method to be used, and the discretization quite independently of each other. To see from the point of view of the users he is addressing see *Physics Today*, special issues on computers in physics, May 1983 and May 1984.

Xxxx82. A speaker from Lawrence Livermore Laboratory (or LASL?), Probably in Fall 1982. A public talk at UC Berkeley, in the Department of Electrical Engineering

and Computer Sciences. In the talk he had explicitly identified the interpretation aspect in computations (as in the upward direction in Figure 4.1 of SNN dissertation).